
Programming for Multibyte Character Sets in LabWindows/CVI

Introduction

This application note discusses how to write your Measurement Studio LabWindows/CVI program so that it is compatible with multibyte character sets.

What is Multibyte?

A traditional character in the C programming language consists of a single byte, which you can set to a particular value from the universal ASCII code. A multibyte character, on the other hand, is a character that can be composed of one or two bytes. A multibyte character set consists of all the multibyte characters required to represent a single language, such as Japanese. A multibyte character composed of only one byte is commonly referred to as a single-byte character. The first byte of a dual-byte character is referred to as the *lead* byte, while the second byte is referred to as the *trail* byte. The *codepage* is the name given to a numeric value that identifies a particular multibyte character set. Refer to the MSDN book, *Developing International Software for Windows 95 and Windows NT*, for more information about multibyte characters sets and international software design in general.

String Handling

The primary rule in manipulating strings that might contain multibyte characters is to always treat the lead byte and the trail byte of a dual-byte character as a single unit. Unfortunately, this affects every instance in your program where characters or strings are handled.

It is important to keep in mind the difference between the length of a string measured in bytes versus the length of a string measured in characters. In many instances, the number of bytes should be used, such as when allocating a buffer for the storage of a string, because every memory storage location of a character needs to allow for the possibility of having a two-byte character. In this case, you should continue to use the ANSI C Library function `strlen`, which returns the number of bytes in a string. In other cases, however, you must replace all ANSI string handling functions with the functions described in the *Multibyte ANSI Extension Functions* section or the macros described in the *Multibyte Macros And Functions In Toolbox.h* section.

Write all of your string processing code in a multibyte-aware manner. For example, pointers should always indicate the start of a character, and indices should always reference the start of a character. Use `CmbStrInc` or `CmbStrDec` instead of the `++` and `--` operators to modify the value of pointers into your strings.

The following is a code example that performs a text search beginning at the end of a string.

Before multibyte changes, your code might look like the following example:

```
char * CVIFUNC FindFileExtension(const char *pathString)
{
    int index, count=0;
    char *fileName;
    char *terminatorPtr;

    AssertMsg(pathString, "Null pathString parameter passed to FindFileExtension");
    fileName = FindFileName(pathString);
    if ((index = strlen (fileName)) == 0)
        return fileName;

    terminatorPtr = fileName + index;
    for (; index > 1; index--) /* do not bother checking when index is 1 because */
    { /* if the dot is in position 0, it really is not an extension */
        if (fileName[index-1] == '.')
            return &fileName[index];
        count++;
        if (count > MAX_FILE_EXTENSION_LENGTH)
            return terminatorPtr;
    }
    return terminatorPtr;
}
```

After multibyte changes, your code might look like the following example:

```
char * CVIFUNC FindFileExtension(const char *pathString)
{
    int index;
    char *fileName;
    char *ptr, *terminatorPtr;

    AssertMsg(pathString, "Null pathString parameter passed to FindFileExtension");
    fileName = FindFileName(pathString);
    if ((index = strlen (fileName)) == 0)
        return fileName;

    terminatorPtr = fileName + index;
    ptr = CmbStrPrev (fileName, terminatorPtr);
    while (ptr && ((terminatorPtr-ptr) <= MAX_FILE_EXTENSION_LENGTH+1))
    {
        if (*ptr == '.' && ptr != fileName) /* if dot is the first char in filename, */
            return ++ptr; /* it really is not an extension */
        CmbStrDec (fileName, ptr);
    }
    return terminatorPtr;
}
```

Multibyte ANSI Extension Functions

The following table contains multibyte aware versions of the ANSI string handling functions.

<code>_getmbcp</code>	Get Current Code Page
<code>_ismbblead</code>	Get Byte Type
<code>_mbsbtype</code>	Get Byte Type from Context
<code>_mbsdec</code>	Get Previous Character
<code>_mbsinc</code>	Get Next Character
<code>_mbslen</code>	Get String Length
<code>_mbncmp</code>	Compare Strings
<code>_mbsncmp</code>	Compare Characters
<code>_mbsicmp</code>	Compare Strings (no case)
<code>_mbsnicmp</code>	Compare Characters (no case)
<code>_mbscat</code>	Concatenate Strings
<code>_mbsncat</code>	Concatenate Characters
<code>_mbscopy</code>	Copy String
<code>_mbsncpy</code>	Copy Characters
<code>_mbschr</code>	Find First Occurrence of Character
<code>_mbsrchr</code>	Find Last Occurrence of Character
<code>_mbspbrk</code>	Find Character from Set
<code>_mbscspn</code>	Find Character from Set (index)
<code>_mbsspn</code>	Find Character Not in Set (index)
<code>_mbsstr</code>	Find Substring
<code>_mbstok</code>	Break String into Tokens

Multibyte Macros And Functions In Toolbox.h

The macros and functions in the following table offer generally useful multibyte-aware functionality that is comparable to the string handling functions in the ANSI C library and pointer arithmetic, such as `*s++`. In these macro and function descriptions, `p` refers to a pointer to the beginning of the string, and `s` refers to the character pointer that you want to move.

<code>OnMBSystem</code>	Returns TRUE if the program is running on a multibyte system.
<code>CmbIsSingleC</code>	Checks if the given 16-bit character fits into a single byte.
<code>CmbGetNumBytesInChar</code>	Returns the number of bytes required to store the given character.
<code>CmbIsLeadByte</code>	Checks if the given byte is a valid lead byte. Call <code>CmbIsLeadByte</code> only if you know that the byte starts on a character boundary; that is, the byte cannot be a trail byte.
<code>CmbCharCodeLeadByte</code>	Returns the lead byte of a dual-byte character.
<code>CmbCharCodeTrailByte</code>	Returns the trail byte of a dual-byte character.
<code>CmbStrByteType</code>	Returns the type of the byte at a given offset within a string, taking into account the bytes before the given byte. If you know that <code>p[offset]</code> is not in the middle of a dual-byte character, you can call <code>CmbIsLeadByte</code> instead. Possible return values are <code>CMB_SINGLE_BYTE</code> , <code>CMB_LEAD_BYTE</code> , <code>CMB_TRAIL_BYTE</code> , and <code>CMB_ILLEGAL_BYTE</code> .
<code>CmbStrDec</code>	Changes <code>s</code> by moving it back by one character. Analogous to <code>--s</code> .
<code>CmbStrInc</code>	Changes <code>s</code> by moving it forward by one character. Analogous to <code>++s</code> .
<code>CmbStrPrev</code>	Returns a pointer to the previous character in the string. Analogous to <code>s-1</code> .
<code>CmbStrNext</code>	Returns a pointer to the next character in the string. Analogous to <code>s+1</code> .
<code>CmbGetC</code>	Retrieves the character at the given pointer. Analogous to <code>*s</code> .
<code>CmbSetC</code>	Sets the character at the given pointer. <code>CmbSetC</code> overwrites any values at the given location and does not properly insert a dual-byte character into the middle of a string. Analogous to <code>*s = c</code> .
<code>CmbGetCInc</code>	First retrieves the character at the given position and then advances the pointer to the next character. Analogous to <code>*s++</code> .
<code>CmbGetCNdxInc</code>	First retrieves the character at the given position and then advances the index to the next character. Analogous to <code>s[i++]</code> .
<code>CmbSetCInc</code>	First sets the character at the given position and then advances the pointer to the next character. The newly set character is returned. Analogous to <code>*s++ = c</code> .
<code>CmbSetCNdxInc</code>	First sets the character at the given position and then advances the index to the next character. The newly set character is returned. Analogous to <code>s[i++] = c</code> .
<code>CmbIncGetC</code>	First advances the given pointer to the next character and then returns that character. Analogous to <code>*++s</code> .
<code>CmbIncSetC</code>	First advances the given pointer to the next character and then sets and returns the new character. Analogous to <code>*++s = c</code> .
<code>CmbFirstByteOfChar</code>	Returns the first byte of the given single or dual-byte character.
<code>CmbNumChars</code>	Returns the number of characters in the given string.

<code>CmbStrEq</code>	Compares the two given strings. Returns <code>TRUE</code> if they are equal.
<code>CmbStrEqI</code>	Compares the two given strings. The comparison is not case-sensitive. Returns <code>TRUE</code> if they are equal.
<code>CmbStrEqN</code>	Compares the first <code>n</code> bytes of the two given strings. Returns <code>TRUE</code> if they are equal.
<code>CmbStrEqNI</code>	Compares the first <code>n</code> bytes of the two given strings. The comparison is not case-sensitive. Returns <code>TRUE</code> if they are equal.
<code>CmbStrCmp</code>	Equivalent to <code>_mbscmp</code> .
<code>CmbStrNCmp</code>	Equivalent to <code>_mbsnbcmp</code> .
<code>CmbStrICmp</code>	Equivalent to <code>_mbsicmp</code> .
<code>CmbStrNICmp</code>	Equivalent to <code>_mbsnbicmp</code> .
<code>CmbStrCat</code>	Equivalent to <code>strcat</code> .
<code>CmbStrNCat</code>	Equivalent to <code>_mbsnbcats</code> .
<code>CmbStrCpy</code>	Equivalent to <code>strcpy</code> .
<code>CmbStrNCpy</code>	Equivalent to <code>_mbsnbcpy</code> .
<code>CmbStrSpn</code>	Equivalent to <code>_mbsspn</code> .
<code>CmbStrCSpn</code>	Equivalent to <code>_mbscspn</code> .
<code>CmbStrChr</code>	Equivalent to <code>_mbschr</code> .
<code>CmbStrRChr</code>	Equivalent to <code>_mbsrchr</code> .
<code>CmbStrTok</code>	Equivalent to <code>_mbstok</code> .
<code>CmbStrPBrk</code>	Equivalent to <code>_mbspbrk</code> .
<code>CmbStrStr</code>	Equivalent to <code>_mbsstr</code> .
<code>CmbStrUpr</code>	Converts all the lowercase characters in the given string to uppercase characters. Converts only English characters.
<code>CmbStrLwr</code>	Converts all the uppercase characters in the given string to lowercase characters. Converts only English characters.
<code>CmbStrByteIs</code>	Returns <code>TRUE</code> if the character at the given offset in the given string is equal to the given single byte character. Analogous to <code>(s[offset] == '*')</code> .
<code>CmbStrLastChar</code>	Returns a pointer to the last character of the given string.

KeyPress Event Handling

KeyPress events are sent to the callback function associated with the control that has the input focus and to the callback function associated with the control's panel.

If your user enters a dual-byte character in a control, two `EVENT_KEYPRESS` events are sent to the callback functions. To process the character as a single event, ignore the first event. You can use `KeyPressEventIsLeadByte` to determine when you receive the first event of a dual-byte character. When you receive the second event, `KeyPressEventIsTrailByte` returns `TRUE`. You can then use `GetKeyPressEventCharacter` to obtain the full dual-byte character.

You can use `SetKeyPressEventKey` to change the character that the user entered. Do not use `SetKeyPressEventKey` in the first `EVENT_KEYPRESS` event of a dual-byte character.

The following is a code example from the user interface sample program, `multikey.prj`.

```
int CVICALLBACK KeyCallback (int panel, int control, int event,
                             void *callbackData, int eventData1, int eventData2)
{
    int character, virtual, modifiers, enabled;
    char buffer[3];
    if (event == EVENT_KEYPRESS) {
        /* remove previous keypress from string control */
        SetCtrlAttribute (panel, PANEL_KEY, ATTR_CTRL_VAL, "");
        /* ignore leading bytes - wait for complete character */
        if (!KeyPressEventIsLeadByte (eventData2)) {
            /* obtain keypress information */
            character = GetKeyPressEventCharacter (eventData2);
            virtual = GetKeyPressEventVirtualKey (eventData2);
            modifiers = GetKeyPressEventModifiers (eventData2);
            /* put the character into a buffer for display */
            memset (buffer, 0, sizeof (buffer));
            CmbSetC (buffer, character);
            /* update the original keypress information */
            SetCtrlAttribute (panel, PANEL_ORIGINAL, ATTR_CTRL_VAL, buffer);
            SetCtrlAttribute (panel, PANEL_CHARACTER, ATTR_CTRL_VAL, character);
            SetCtrlAttribute (panel, PANEL_VIRTUAL, ATTR_CTRL_VAL, virtual);
            SetCtrlAttribute (panel, PANEL_MODIFIERS, ATTR_CTRL_VAL, modifiers >> 16);
            /* is filtering enabled */
            GetCtrlVal (panel, PANEL_FILTER, &enabled);
            if (enabled) {
                /* get filter information */
                GetCtrlVal (panel, PANEL_CHAR_FILTER, &character);
                GetCtrlVal (panel, PANEL_VIRT_FILTER, &virtual);
                GetCtrlVal (panel, PANEL_MODI_FILTER, &modifiers);
                /* replace key event */
                SetKeyPressEventKey (eventData2, virtual, character, modifiers
                                    << 16, 0, virtual == 0 && !CmbIsSingleC (character));
            }
        }
    }
    return 0;
}
```

