Output Variables

The ConvertGlobalStringTextEncoding operation returns information in the following variables:

`V_numConversionsSucceeded`	V_numConversionsSucceeded is set to the number of successful text conversions.
`V_numConversionsFailed`	V_numConversionsFailed is set to the number of unsuccessful text conversions.
`V_numConversionsSkipped`	V_numConversionsSkipped is set to the number of skipped text conversions. Text conversion is skipped if the string variable contains binary data and the /SKIP=0 flag is omitted. Text conversion is skipped by default if the string variable contains binary data. Text conversion is also skipped by default if *newTextEncoding* is 1 (UTF-8) and the string variable is already valid as UTF-8. See the /SKIP flag for details.

Examples

```
// In the following examples 1 means UTF-8, 4 means Shift JIS.

// Convert specific strings' content from Shift JIS to UTF-8
ConvertGlobalStringTextEncoding 4, 1, string0, string1

// Convert all strings' content from Shift JIS to UTF-8
ConvertGlobalStringTextEncoding /DF={root:,1} 4, 1

// Same as before but exclude the root:Packages data folder
ConvertGlobalStringTextEncoding /DF={root:,1,root:Packages} 4, 1

// Convert all strings’ content from Shift JIS to UTF-8 except strings containing binary
ConvertGlobalStringTextEncoding /DF={root:,1}/SKIP=0 4, 1
```

See Also

Text Encodings on page III-459, **String Variable Text Encodings** on page III-478, **Text Encoding Names and Codes** on page III-490

ConvertTextEncoding, **SetWaveTextEncoding**

ConvertTextEncoding

`ConvertTextEncoding(`*`sourceTextStr`*`, `*`sourceTextEncoding`*`, `*`destTextEncoding`*`, `*`mapErrorMode`*`, `*`options`*`)`

ConvertTextEncoding converts text from one text encoding to another.

The ConvertTextEncoding function was added in Igor Pro 7.00.

All text in memory is assumed to be in UTF-8 format except for text stored in waves which can be stored in any text encoding. You might want to convert text from UTF-8 to Windows-1252 (Windows Western European), for example, to export it to a program that expects Windows-1252.

You might have text already loaded into Igor that you know to be in Windows-1252. To display it correctly, you need to convert it to UTF-8.

You can also use ConvertTextEncoding to test if text is valid in a given text encoding, by specifying the same text encoding for *sourceTextEncoding* and *destTextEncoding*.

Parameters

sourceTextStr is the text that you want to convert.

sourceTextEncoding specifies the source text encoding.

destTextEncoding specifies the output text encoding.

See **Text Encoding Names and Codes** on page III-490 for a list of acceptable text encoding codes.

mapErrorMode determines what happens if an input character can not be mapped to the output text encoding because the character does not exist in the output text encoding. It takes one of these values:

options is a bitwise parameter which defaults to 0 and with the bits defined as follows:

1: Generate error. The function returns "" and generates an error.

2: Return a substitute character for the unmappable character. The substitute character for Unicode is the Unicode replacement character, U+FFFD. For most non-Unicode text encodings it is either control-Z or a question mark.

3: Skip unmappable input character.

4: Return escape sequences representing unmappable characters and invalid source text.

If the source text is valid in the source text encoding but can not be represented in the destination text encoding, unmappable characters are replaced with \uXXXX where XXXX specifies the UTF-16 code point of the unmappable character in hexadecimal. The DemoUnmappable example function below illustrates this.

If the conversion can not be done because the source text is not valid in the source text encoding, invalid bytes are replaced with \xXX where XX specifies the value of the invalid byte in hexadecimal. The DemoInvalid example function below illustrates this.

If mapErrorMode is 2, 3 or 4, the function does not return an error in the event of an unmappable character.

Bit 0: If cleared, in the event of a text conversion error, a null string is returned and an error is generated. Use this if you want to abort procedure execution if an error occurs.

If set, in the event of a text conversion error, a null string is returned but no error is generated. Use this if you want to detect and handle a text conversion error. You can test for null using strlen as shown in the example below.

Bit 1: If cleared (default), null bytes in *sourceTextStr* are considered invalid and ConvertTextEncoding returns an error. If set, null bytes are considered valid.

Bit 2: If cleared (default) and *sourceTextEncoding* and *destTextEncoding* are the same, ConvertTextEncoding attempts to do the conversion anyway. If *sourceTextStr* is invalid in the specified text encoding, the issue is handled according to *mapErrorMode*. This allows you to check the validity of text whose text encoding you think you know, by passing 1 for *mapErrorMode* and 5 for *options*. Use **strlen** to test if the returned string is null, indicating that *sourceTextStr* is not valid in the specified text encoding.

If set and *sourceTextEncoding* and *destTextEncoding* are the same, ConvertTextEncoding merely returns *sourceTextStr* without doing any conversion.

All other bits are reserved and must be cleared.

Details

ConvertTextEncoding returns a null result string if *sourceTextEncoding* or *destTextEncoding* are not valid text encoding codes or if a text conversion error occurs. You can test for a null string using strlen which returns NaN if the string is null.

If bit 0 of the *options* parameter is cleared, Igor generates an error which halts procedure execution. If it is set, Igor generates no error and you should test for null and attempt to handle the error, as illustrated by the example below.

A text conversion error occurs if *mapErrorMode* is 1 and the source text contains one or more characters that are not mappable to the destination text encoding. A text conversion error also occurs if the source text contains a sequence of bytes that is not valid in the source text encoding.

The "binary" text encoding (255) is not a real text encoding. If either *sourceTextEncoding* or *destTextEncoding* are binary (255), ConvertTextEncoding does no conversion and just returns *sourceTextStr* unchanged.

See **Text Encodings** on page III-459 for further details.

Example

In reading these examples, keep in mind that Igor converts escape codes such as “\u8C4A”, when they appear in literal text, to the corresponding UTF-8 characters. See **Unicode Escape Sequences in Strings** on page IV-15 for details.

```
Function DemoConvertTextEncoding()
    // Get text encoding codes for text the text encodings used below
    Variable teUTF8 = TextEncodingCode("UTF-8")
    Variable teWindows1252 = TextEncodingCode("Windows-1252")
    Variable teShiftJIS = TextEncodingCode("ShiftJIS")

    // Convert from Windows-1252 to UTF-8
    String source = "Division sign: " + num2char(0xF7)
    String result = ConvertTextEncoding(source, teWindows1252, teUTF8, 1, 0)
    Print result

    // Convert unmappable character from UTF-8 to Windows-1252
    // \u8C4A is an escape sequence representing a Japanese character in Unicode
    // for which there is no corresponding character in Windows-1252

    // Demonstrate mapErrorMode = 1 (fail unmappable character)
    source = "Unmappable character causes failure: {\u8C4A}"
    // Pass 1 for options parameter to tell Igor to ignore error and let us handle it
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 1, 1)
    Variable len = strlen(result)     // Will be NaN if conversion failed
    if (NumType(len) == 2)
        Print "Conversion failed (as expected). Result is NULL."
        // You could cope with this error by trying again with the mapErrorMode
        // parameter set to 2, 3 or 4.
    else
        // We should not get here
        Print "Conversion succeeded (should not happen)."
        Print result
    endif

    // Demonstrate mapErrorMode = 2 (substitute for unmappable character)
    source = "Unmappable character replaced by question mark: {\u8C4A}"
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 2, 0)
    Print result                      // Prints "?" in place of unmappable character

    // Demonstrate mapErrorMode = 3 (skip unmappable character)
    source = "Unmappable character skipped: {\u8C4A}"
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 3, 0)
    Print result                      // Skips unmappable character

    // Demonstrate mapErrorMode = 4 (insert escape sequence for unmappable character)
    source = "Unmappable character replaced by escape sequence: {\u8C4A}"
    result = ConvertTextEncoding(source, teUTF8, teWindows1252, 4, 0)
    Print result        // Unmappable character represented as escape sequence

    // Demonstrate mapErrorMode = 4 (insert escape sequence for unmappable character)
    source = "Unmappable character replaced by escape sequence: {\u8C4A}"
    // First convert UTF-8 to Shift_JIS (Japanese). This will succeed.
    result = ConvertTextEncoding(source, teUTF8, teShiftJIS, 1, 0)
    // Next convert Shift_JIS (Japanese) to Windows-1252. The character can not
    // be mapped and is replaced by an escape sequence.
    result = ConvertTextEncoding(result, teShiftJIS, teWindows1252, 4, 0)
    Print result        // Unmappable character represented as escape sequence
End

// Demo unmappable character
// In this example, the source text is valid but not representable in destination
// text encoding. Because we pass 4 for the mapErrorMode parameter, ConvertTextEncoding
// uses an escape sequenceto represent the unmappable text.
Function DemoUnmappable()
    String input = "\u2135" // Alef symbol - available in UTF-8 but not in MacRoman
    String output = ConvertTextEncoding(input, 1, 2, 4, 0)
    Print output            // Prints "\u2135"
End

// Demo invalid input text
// In this example, the source text is invalid in the source text encoding.
// Because we pass 4 for the mapErrorMode parameter, ConvertTextEncoding uses an escape
    sequences
// to represent the invalid text.
Function DemoInvalidInput()
    String input = "\x8E"   // Represents "é" in MacRoman but is not valid in UTF-8
    String output = ConvertTextEncoding(input, 1, 2, 4, 0)
    Print output            // Prints "\x8E"
End
```