# Character Encoding

Overview of character encodings used in Unreal Engine.

This document provides an overview of character encodings used in **Unreal Engine (UE)**.

Assumed knowledge: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

## Text Formats

There are several formats that can be used to represent text and strings. Understanding these formats and their inherent pros and cons can help in making decisions on what formats to use in your projects.

These are not the technical definitions of formats, but rather simplified versions suitable for this page.

$ **ASCII** : Characters between 32 and 126 inclusive, and 0, 9, 10, and 13. (P4 type text) (This is validated with a P4 trigger on check-in) $ **ANSI** : ASCII and the current codepage (e.g. Western European high ASCII) needs to be stored as binary on the P4 server. $ **UTF-8** : A string made up of single bytes which can use special character sequences to get non-ANSI characters. (a superset of ASCII) (P4 type Unicode) $ **UTF-16** : A string made up of 2 bytes per character with a BOM. (although can go to 4 bytes with astral characters) (P4 type UTF-16) (This is validated with a P4 trigger on check-in)

## The Case for Binary

| Pros | Cons |
|---|---|
| Internal format is not defined; each file can be loaded no matter what format it is. | Non mergable. Requires all files of this type to be exclusive checkout. |
| | Internal format is not defined; each file could be in a different format. |
| | P4 stores the entirety of each version, which can unnecessarily bloat the depot size. |

## The Case for Text

| Pros | Cons |
|---|---|
| Merge-able. Exclusive checkout is not required. | Very limiting; only ASCII characters allowed. |

## The Case for UTF-8

| Pros | Cons |
|---|---|
| Simple access to all characters we will ever need. | Has a different memory profile for Asian languages. |
| Uses less memory. | P4 type Unicode is not enabled on our Perforce server. |

| Pros | Cons |
|---|---|
| Is a superset of ASCII; a plain ASCII string is a perfectly valid UTF-8 string. | String operations more complicated; have to parse the string to do something as simple as a length calculation. |
| Still works when the game detects the string is ASCII and outputs it as such. | MSDev does not handle anything other than ASCII very well in Asian regions. This is why we validate text as ASCII during check-in. |
| If we did have a Unicode enabled server, the files would be merge-able and exclusive checkout would not be required. | |
| Can detect whether a string is UTF-8 by parsing it (with or without a BOM). | |

## The Case for UTF-16

| Pros | Cons |
|---|---|
| Simple access to all characters we will ever need. | Uses more memory. |
| Simple. Memory usage is twice the number of characters (for characters we use, which are all in the Basic Multilingual Plane). | Difficult to detect this format if it does not have a BOM. |
| Simple. String operations can split/combine without having to parse the strings. | Does not work when the game detects the string is ASCII and outputs it as such (this is now detected on check-in with the UTF-16 validator). |
| Same as the format used in game, no translation, parsing or memory operations required. | MSDev does not handle anything other than ASCII very well in Asian regions. This is why we validate text as ASCII during check-in. |

| Pros | Cons |
|---|---|
| Merge-able. Exclusive checkout is not required. | |
| C# uses UTF-16 internally. | |

## UE Internal String Representation

All strings in UE are stored in memory in UTF-16 format as FStrings or TCHAR arrays. Most code assumes 2 bytes is one codepoint so only the Basic Multilingual Plane (BMP) is supported so Unreal's internal encoding is more correctly described as UCS-2. Strings are stored in the endian-ness appropriate for the current platform.

When serializing to packages to/from disk or during networking, strings with all TCHAR characters less than 0xff are stored as a series of 8-bit bytes, and otherwise as 2-byte UTF-16 strings. The serialization code can deal with any endian conversion as necessary.

## Text Files Loaded by UE

When UE loads an external text file (for example reading a `.INT` file at runtime) it is almost always done with the `appLoadFileToString()` function found in `UnMisc.cpp`. The main work occurs in the `appBufferToString()` function.

This function recognizes the Unicode byte-order-mark (BOM) in a UTF-16 file and if present, will load the file as UTF-16 in either endian-ness.

What happens when the BOM is not present depends on the platform.

On Windows, it will attempt to convert the text to UTF-16 using the default Windows MBCS encoding (eg Windows-1252 for US English and Western Europe, CP949 for Korean and CP932 for Japanese) and uses MultiByteToWideChar(CP_ACP, MB_ERR_INVALID_CHARS...). This was added around the July 2009 QA build.

If this conversion fails on platforms other than Windows, it will just read each byte and pad it to 16-bits to make an array of `TCHAR`.

Note that there is no code to detect or decode UTF-8 encoded text files loaded with `appLoadFileToString()`.

## Text Files Saved by UE

Most text files generated by the engine will be saved using `appSaveStringToFile()`.

Strings with all TCHAR characters representable by a single byte will be stored as a series of 8-bit bytes, and otherwise as UTF-16 unless the bAlwaysSaveAsAnsi flag is passed in as true, in which case it will be converted to the default Windows encoding first. This is currently only done on shader files, to work around an issue a shader compiler had with UTF-16 files.

## Recommended Encoding for Text Files Used by UE

### INT and INI Files

UTF-16 in either endian. While the default MBCS encoding for an Asian language (eg CP932) will work on Windows, these files need to be loaded on PS3 and Xbox360 and the conversion code only runs on Windows.

### Source Code

In general, we do not recommend string literals inside C++ source code and we recommend this data goes in INT files.

#### C++ Source Code

UTF-8 or default Windows encoding. MSVC, the Xbox360 compiler and gcc should all be happy with UTF-8 encoded source files. Latin-1 encoded files with characters with the high bit set, for example copyright, trademark or degree symbols should be avoided in source code where possible because the encoding will break on systems with different locales. Some instances of this in 3rd party software are unavoidable (eg copyright notices) so for

MSVC we disable warning 4819, which would otherwise occur when compiling on Asian Windows.

## Storing UTF-16 Text Files in Perforce

- Do not use 'Text'
  - If a UTF-x file is checked in and stored as text, it will be corrupted after syncing.
- If you use 'Binary', mark the files as exclusive checkout
  - People can check in ASCII, UTF-8, UTF-16 and it will work in engine.
  - However, binary files cannot be merged, so if the files are not marked as exclusive checkout, changes will be stomped upon.
- If you use 'UTF-16', make sure no one checks in a file that is not UTF-16
- The 'Unicode' type is UTF-8, and of no use to us here.

## Conversion Routines

We have a number of macros to convert strings to and from various encodings. These macros use a class instance declared in local scope and allocate space on the stack, so it is very important you do not retain pointers to them! They are intended only for passing strings to function calls.

- TCHAR_TO_ANSI(str)
- TCHAR_TO_OEM(str)
- ANSI_TO_TCHAR(str)
- TCHAR_TO_UTF8(str)
- UTF8_TO_TCHAR(str)

These use the following helper classes from UnStringConv.h:

- `typedef TStringConversion<TCHAR,ANSICHAR,FANSIToTCHAR_Convert> FANSIToTCHAR;`
- `typedef TStringConversion<ANSICHAR,TCHAR,FTCHARToANSI_Convert> FTCHARToANSI;`
- `typedef TStringConversion<ANSICHAR,TCHAR,FTCHARToOEM_Convert> FTCHARToOEM;`
- `typedef TStringConversion<ANSICHAR,TCHAR,FTCHARToUTF8_Convert> FTCHARToUTF8;`
- `typedef TStringConversion<TCHAR,ANSICHAR,FUTF8ToTCHAR_Convert> FUTF8ToTCHAR;`

It is also critical that when using TCHAR_TO_ANSI you do not assume the number of bytes will be the same as the TCHAR string length. Multiple byte character sets could require multiple bytes per TCHAR character. If you need to know the length of the resulting string in bytes, you can use the helper class instead of the macros. For example:

```
FString String;
...
FTCHARToANSI Convert(*String);
Ar->Serialize((ANSICHAR*)Convert.Get(), Convert.Length()); // FTCHARToANSI::Length() returns the number of bytes for the encoded string, excluding the null terminator.

```

Copy full snippet

The objects that these macros declare have very short lifetimes. The intended use case for them is as function parameters, and they are suited to this situation. Do not assign a variable to the contents of the converted string, as the object will go out of scope and the string will be released. This can cause crashes if your code continues to access the pointer to released memory.

## ToUpper() and ToLower() Non-Trivial in Unicode

UE4 currently only handles ANSI (ASCII | code page 1252 | | Western European).

The least worst method to do this for all languages seems to be mentioned here

- ISO/IEC 8859-1 for English, French, German, Italian, Portuguese, and both Spanishes
- ISO/IEC 8859-2 for Polish, Czech, and Hungarian
- ISO/IEC 8859-5 for Russian

The mappings from unicode.org contain the conversion rules for the above mentioned languages. 'CAPITAL LETTER' and 'SMALL LETTER' info would be cross referenced with the appropriate unicode character to get the desired result.

# Notes about C++ Source Code Specific to East Asian Encodings

Both UTF-8 and the default Windows encodings can cause problems with the C++ compiler, as follows:

**Default Windows encoding**

Take care when compiling C++ source code on Windows with running with a single byte character code page (e.g. CP437 United States), if the source code has an East Asian double byte character encoding such as CP932 (Japanese), CP936 (Simplified Chinese), or CP950 (Traditional Chinese).

These East Asian character encoding systems uses 0×81-0xFE for first byte, and 0×40-0xFE for second byte. A value of 0×5C in the second byte will be interpreted as backslash in ASCII/latin-1, and that has a special meaning for C++. (Escape sequence inside a string literal, and line continuation if used at the end of a line). When compiling that source code on single-byte code page Windows, the compiler does not care about the East Asian double byte character encoding, and this could cause either a compile error or worse, create a bug in the EXE.

Single-line comments: These can cause difficult-to-find bugs or errors caused by a missing line, if in the end of East Asian comment has 0×5c.

```
// EastAsianCharacterCommentThatContains0x5cInTheEndOfComment0x5c'\'
important_function(); /* this line would be connected to above line as part of comment */
```

Copy full snippet

Inside a string literal: This can cause a broken string or an error with an recognized 0×5c escape sequence.

```
printf("EastAsianCharacterThatContains0x5c'\'AndIfContains0x5cInTheEndOfString
function();
printf("Compiler recognizes left double quotation mark in this line as the end
literal that continued from first line, and expected this message is C++ code.
```

Copy full snippet

If the character following 0×5c does specify a escape sequence, compiler converts escape sequence character set to single specified character. (If does not specify, the result is implementation defined, but MSVC removes 0×5c, and warns "unrecognized character escape sequence".) In the above case, the end of string has a 0×5c backslash and next character is a double quote, so the escape sequence " is converted to a double quote in the string data, and compiler continues to make string data before next double quote or end of file, and causes an error.

Examples of dangerous characters: CP932 (Japanese Shift-JIS) "?" is 0×955C, and so many CP932 characters have 0×5C. CP936 (Simplified Chinese GBK) "?" is 0×815C, and so many CP936 characters have 0×5C. CP950 (Traditional Chinese Big5) "?" is 0xA55C, and so many CP950 characters have 0×5C. CP949 (Korean, EUC-KR) is OK, because EUC-KR does not use 0×5C for the second byte.

**UTF-8 without BOM** (Some text editors describe BOM as signature)

Take care for compiling C++ source code on East Asian code page CP949 (Korean), CP932 (Japanese), CP936 (Simplified Chinese) or CP950 (Traditional Chinese) Windows, if that source code has an East Asian character stored as UTF-8.

UTF-8 character encoding uses three bytes for East Asian characters: 0xE0-0xEF for the first byte, 0×80-0xBF for the second byte and 0×80-0xBF for the third byte. Without the BOM, East Asian Windows' default encoding recognizes the three UTF-8 encoded bytes and the following byte as two 2-byte East Asian encoded characters, pairing of first and second bytes for one first East Asian character, and the third byte and following byte paired to form the second East Asian character. Problems can occur if the character following the UTF-8 encoded three bytes has special meaning in the string literals or comments.

Eg In in-line comment: Causes hard-to-find bugs or errors with missing code, if the comment text contains an odd number of East Asian characters, and next character marks the end of the comment.

```
/*OddNumberOfEastAsianCharacterComment*/
important_function();
/*normal comment*/
```

Copy full snippet

The compiler on East Asian code page Windows recognizes the last byte of the UTF-8 decoded East Asian character comment and asterisk * as a single East Asian character, and next characters is treated as still part of the comment. In above case, compiler removes important_function() as it seems to be part of the comment. This behavior is very dangerous and it is difficult to find the missing code.

In single-line comment: Using backslash '' at the end of an East Asian comment causes hard-to-find bugs or errors without missing lines.

```
// OddNumberOfEastAsianCharacterComment\
description(); /* coder intended this line as comment, by using backslash at the end of above line */
```

Copy full snippet

This is a very rare case, because programmers should not intentionally write backslashes '' at the end of comments.

Inside string literals: This causes broken strings, errors or warnings when an odd number of UTF-8 encoded East Asian characters are inside a string literal and the following character has special meaning.

```
printf("OddNumberOfEastAsiaCharacterString");
printf("OddNumberOfEastAsiaCharacterString%d",0);
printf("OddNumberOfEastAsiaCharacterString\n");
```

Copy full snippet

The C++ compiler on East Asian code page Windows interprets the last byte of the UTF-8 decoded East Asian character string and next character as a single East Asian character. If you are lucky, compiler warning "C4819" (if not disabled) or an error will alert you to the problem. If unlucky, the string would be broken.

**Conclusion**

You can use UTF-8 or default Windows encoding for C++ source code, but please be aware of these problem. Again, we do not recommend string literals inside C++ source. Please make sure to use East Asian as your default code page if you have to use East Asian character encoding in C++ source code. Another good way is to use UTF-8 with BOM (some text editors describes the BOM as a Unicode signature).

We tested a few compilers with UTF-8 and UTF-16 on 18 Feb 2010.

MSVC for PC and Xbox 360, and gcc or slc for PS3 are able to compile UTF-8-encoded source code (with and without BOM). But UTF-16 (little-endian/big-endian) is supported only by MSVC.

Perforce is able to work with both UTF-16 and UTF-8, but p4 diff displays the BOM in UTF-8 files as a visible character.

External reference: Code Pages Supported by Windows