# 1 What's New in the FDK

This chapter lists changes to the Frame® Developer's Kit (FDK) that are the result of changes to the Adobe® FrameMaker® 8 product release.

## What's New for Release 8

FrameMaker 8 introduced the following changes that affect the FDK:

- Unicode support
- Use of International Components for Unicode (ICU) in the FDK
- FrameMaker Product Activation
- Support for 3D
- Support for Adobe Flash®
- Enhancement of the RTF filter
- New APIs
- New or changed FDK properties
- Use of Microsoft® Visual Studio® 2005 to compile

## Unicode Support in FrameMaker

Because 8 and later versions of FrameMaker provide full Unicode support, all FDK APIs and FDE functions have been modified to handle Unicode when running in *Unicode Mode*. When running in this mode, all FDK APIs and FDE functions require string data in all input arguments to be in Unicode. All return values from (and parameters set by) FDK APIs and FDE functions in this mode will also be in Unicode. FDE string utility functions that explicitly require string encoding as a parameter or have been made for a specific encoding are an exception to these rules.

Except where specifically mentioned, all APIs expect Unicode data to be in UTF-8 encoding, rather than UTF-16 or UTF-32. Providing invalid Unicode input to APIs in *Unicode Mode* will cause unpredictable behavior. A new API, `F_IsValidUTF8`, ascertains whether a string is valid in the UTF-8 encoding. The client must ensure that all input to an API is in the valid encoding.

When FDK APIs and the FDE aren't running in *Unicode Mode*, they running in *Compatibility Mode*. This is the default mode of operation and provides backward compatibility for legacy

clients that can't handle Unicode or/and have not been compiled with FDK 8. A detailed description of *Compatibility Mode* can be found in a later section.

## Enable *Unicode Mode* in the FDK

*Unicode Mode* must be enabled separately for the two principal parts of the FDK: APIs and the FDE.

### Enable *Unicode Mode* for APIs

To enable *Unicode Mode* for APIs, call the new API `F_ApiEnableUnicode` as follows:

```
F_ApiEnableUnicode(True);
```

Make this call before calling any other API except the APIs required for setting up communication for asynchronous clients. Any API call made before this one won't work in *Unicode Mode*. Calling this multiple times is safe, but only once is necessary for clients that do not unload. For clients that do unload, this must be called every time the client is reloaded. For this reason, you should make this the first call in `F_ApiInitialize` of the client.

The behavior of the callback `F_ApiInitialize` has also changed. It will now be called for all API clients including filters and document reports. Filters can enable *Unicode Mode* in `F_ApiInitialize` before receiving a call to the callback `F_ApiNotify` and thus receive `sparm` in Unicode.

You can subsequently disable *Unicode Mode* by making the following call:

```
F_ApiEnableUnicode(False);
```

### Enable Unicode Mode for the FDE

*Unicode Mode* for FDE functions is enabled if the encoding of the FDE has been set to UTF-8 using the function `F_FdeInitFontEncs`. A new encoding, `"UTF-8"`, has been added as a possible input value for `F_FdeInitFontEncs`. This can be called as follows:

```
FontEncIdT feId;
feId = F_FdeInitFontEncs((ConStringT) "UTF-8");
```

Or, simply as:

```
F_FdeInitFontEncs((ConStringT) "UTF-8");
```

Make this call before calling any other FDE function. Any FDE function called before this won't work in *Unicode Mode*. It is safe to call this function more than once. Calling `F_FdeInitFontEncs` with an encoding other than `"UTF-8"` disables *Unicode Mode* for the FDE.

### Mixed Mode operations

Using the calls mentioned above, you can run APIs in *Unicode Mode* while running the FDE in *Compatibility Mode*, and vice versa. The resulting mode is called a *Mixed Mode*. You can

also keep switching between *Unicode Mode* and *Compatibility Mode*. This mode is also called a *Mixed Mode* because the client runs in different modes at different points in time.

Such *Mixed Mode* operations are potentially hazardous because they may result in inconsistently encoded strings. For example, a filepath enumeration code created using the FDE function `F_FilePathGetNext` returns the filenames in UTF-8 if the FDE is running in *Unicode Mode*. However, if FDK APIs are running in *Compatibility Mode*, `F_ApiOpen` can't open the filename provided by `F_FilePathGetNext` because it can't handle Unicode in *Compatibility Mode*.

**IMPORTANT:** *Mixed Mode operations aren't recommended and can result in unpredictable behavior. FDK API and FDE modes must be changed together and before making calls to any other APIs or FDE functions (with the exception of F_SetICUDataDir and F_GetICUDataDir).*

## Behavior of FDK APIs in *Unicode Mode*

In *Unicode Mode*, all APIs require all strings in input parameters to be valid UTF-8 strings. Strings in all return values or parameters set by APIs in this mode will be in UTF-8. For example, `F_ApiAlert` accepts UTF-8 input in *Unicode Mode:*

```
F_ApiEnableUnicode(True);
F_ApiAlert("This is Unicode: \xC3\xA4 \xEB\xAE\xA4 \xD8\xB4",
     FF_ALERT_CONTINUE_NOTE);
```

where `0xC3 0xA4` is the UTF-8 representation of ä (Latin), `0xEB 0xAE 0xA4` of 뮤 (Hangul), and `0xD8 0xB4` of ش (Arabic). Putting a `\x` followed by the hex code `C3` puts a byte `0xC3` in the string. This is a feature provided by C/C++ compilers and isn't specific to FrameMaker. This behavior might not work for some compilers. The alert produced by this call is shown below:


### F_ApiGetSupportedEncodings

In *Unicode Mode*, the API `F_ApiGetSupportedEncodings` returns a list containing only `"UTF-8"` because this is the only supported encoding in this mode.

### F_ApiIsEncodingSupported

In *Unicode Mode*, `F_ApiIsEncodingSupported` returns `True` only for `"UTF-8"` because this is the only supported encoding in this mode.

### F_ApiSave

In the FDK 8 *Unicode Mode*, `FV_SaveFmtBinary` and `FV_SaveFmtInterchange` are equivalent to `FV_SaveFmtBinary80` and `FV_SaveFmtInterchange80` respectively when calling `F_ApiSave`. Therefore, the default mode of saving in *Unicode Mode* is 8 format (FM8, MIF8, FM Book 8, MIF Book 8)

### F_ApiAddText F_ApiGetText, F_ApiGetText2, F_ApiGetTextForRange, F_ApiGetTextForRange2

These APIs have a peculiar behavior with respect to Symbol/Dingbats/Webdings fonts in *Unicode Mode*. The internal representation of a character 'α' in Symbol (`'a'` with Symbol font applied) is 0x61, which is the same as the representation of `'a'`. This symbol isn't stored as the Greek Unicode letter 'α', which has the Unicode code point `0x03B1`. Therefore, if Symbol font is applied to `'a'`, it turns into 'α'. If a font like Times New Roman is subsequently applied, it turns back into 'a'. The same is true for other characters in Symbol, Dingbats, and Webdings fonts. Hence, if the text "αβχ" (which is actually the text `"abc"` with the Symbol font applied) is obtained using `F_ApiGetText`, the string `"abc"` is returned.

The symbol ∇ , which has the symbol representation of `0xD1` (see the *FrameMaker Character Sets* document) will be represented by the byte sequence `0xC3 0x91`. This byte sequence is the UTF-8 representation of the Unicode code point `0xD1`, which represents the character Ñ in Unicode. Therefore, applying Symbol font to Ñ results in ∇ .

Because of this representation, Symbol/Dingbats/Webdings characters that are obtained by applying the respective fonts to non-ASCII characters are returned as two-byte sequences by these APIs. For example, using `F_ApiGetText` to get the text " ∇ " returns the byte sequence `0xC3 0x91` (or the UTF-8 character Ñ). To convert this to a single byte symbol representation (necessary in FrameMaker 7.2 and earlier), the client must explicitly convert the UTF-8 byte sequence `0xC3 0x91` to the UTF-32 code point 0x00D1 and then store it in a single byte.

The `F_StrConvertEnc` function(s), which can also be used to convert from Symbol/Dingbats/Webdings to UTF-8, expect the input to be in Symbol/Dingbats/Webdings encodings. Therefore, the client must convert the UTF-8 byte sequence to a single byte before sending it to these functions for conversion.

The above behavior is true for other variants of `F_ApiGetText` as well. For example, to add the text ∇ in the document, the UTF-8 byte sequence `0xC3 0x91` must be sent to `F_ApiAddText`. Applying the Symbol font to this added text displays the desired character.

### F_ApiGetEncodingForFont, F_ApiGetEncodingForFamily

The APIs `F_ApiGetEncodingForFont` and `F_ApiGetEncodingForFamily` have the same behavior as in FDK 7.2 in both *Compatibility Mode* and *Unicode Mode*. That is, these do not return `"UTF-8"` as the encoding for any font in either mode.

## Behavior of FDE functions in *Unicode Mode*

In the FDE, functions found under the following headings in the *FDK Programmer's Reference* do not depend on the FDE encoding and, therefore, have the same behavior in *Unicode Mode* and *Compatibility Mode*:

- Characters
- F-Codes
- Fonts
- Hash Tables
- Memory: manipulating with handles
- Memory: manipulating with pointers
- Metrics
- String lists
- Strings: allocating, copying, and deallocating
- Strings: comparing and parsing
- Strings: concatenating
- Strings: miscellaneous
- Strings: encoded

These include functions that either do not deal with strings (for example `F_MetricSqrt`), expect strings/characters to be in FrameRoman encoding (for example, `F_StrReverse`), can work for any encoding other than UTF-16 or UTF-32 (for example, `F_StrCopyString`), or explicitly ask for the encoding of the strings (for example, `F_StrLenEnc`).

FDE functions found under the following headings in *FDK Programmer's Reference* depend on the FDE encoding and have different behaviors in *Unicode Mode* and *Compatibility Mode*. When *Unicode Mode* is enabled for the FDE, these expect all strings in input parameters to be valid UTF-8 strings. Strings in all return values or parameters set by these functions in this mode will be in UTF-8.

### Debugging

The following API functions found in the *FDK Programmer's Reference* under the heading *Debugging* are a notable exception to conventions. These behave as a part of the FDE, rather than as a part of the APIs, and thus depend upon the FDE encoding. Their behavior is similar to the behavior of `F_Printf` (see under the I/O section).

- `F_ApiPrintTextItem`
- `F_ApiPrintTextItems`
- `F_ApiPrintPropVal`
- `F_ApiPrintPropVals`

### Files, directories, and filepaths

These functions can deal with Unicode paths when *Unicode Mode* is enabled. Like everywhere else, all inputs are expected to be in UTF-8 and all outputs are in UTF-8.

The behavior of these functions on Solaris™ is slightly different from that on Windows®. Solaris stores filenames in a byte-based manner, rather than in a specific encoding. Therefore, filenames can be in EUC, ANSI, and UTF-8 while running on a single locale (for example, `en_US.UTF-8)`. However, you might see only the UTF-8 filenames correctly on this locale. In the `ja_JP.EUC` locale, you might see only the EUC filenames correctly.

The FDK 8 *Unicode Mode* provides a method for accessing filenames in UTF-8 irrespective of the Solaris locale. All FDE calls require input parameters to be in UTF-8 encoding while in *Unicode Mode*. These filepaths are internally maintained in UTF-8 and are converted to the native encoding of the Solaris locale before making system calls. Filenames returned by the OS are converted from the native encoding of the locale to UTF-8.

Filenames that aren't valid in the native encoding of the locale get converted to an escape sequence. For example, in *Unicode Mode,* if you use `F_FilePathGetNext` to enumerate the list of files in a directory that contains a filename `"uni.txt"` in UTF-8 encoding, you might get a filepath like `"uni%XC3%XA4.txt"` if the byte sequence `0xC3 0xA4` (representation in UTF-8) isn't valid in the current locale. When such a filepath is provided to a FilePath function like `F_RenameFile`, the function handles the escape sequence by decoding it to a sequence of bytes and is, therefore, able to access the correct file.

Filenames that are valid in the native encoding of the locale get converted without escape sequences. Filenames that are in UTF-8 encoding while on a UTF-8 locale do not need any conversion.

### I/O

The calls `F_ChannelOpen` and `F_ChannelMakeTmp` can accept UTF-8 filenames when in *Unicode Mode*. They handle filepaths in the same manner as FilePath functions discussed above. The calls `F_Printf` and `F_Warning` also require UTF-8 input in *Unicode Mode*.

In *Unicode Mode*, all data written to the `NULL` channel (the console) using `F_Printf`, `F_Warning`, or `F_ChannelWrite` must be in UTF-8. The behavior of these calls, however, differs on Windows and Solaris. On Windows, because the console window can display Unicode irrespective of the locale, the UTF-8 input is displayed identically across all locales.

On Solaris, because the console window can display only byte sequences valid on the current locale, the data to be printed is converted from UTF-8 to the encoding of the current locale (ANSI in `en_US`, UTF-8 in `ja_JP.UTF-8`, and Shift-JIS on `ja`, for example) before being sent to the console. Hence, the characters displayed on the console are limited by the locale in Solaris.

**IMPORTANT:** *Because on Solaris* `F_Printf` *can display Unicode characters properly only if running on a UTF-8 locale, all examples that involve* `F_Printf` *in this document work correctly only on a UTF-8 locale.*

#### Maker Interchange Format (MIF)

These calls can accept UTF-8 input when in *Unicode Mode*. In *Unicode Mode*, `F_MifString` writes all `FM_Tab`, `FM_NonBrkHyphen`, `FM_DiscHyphen`, and `FM_HardSpace` in a string by starting a separate character tag. So a string `"space-time"` with a nonbreaking hyphen between `"space"` and `"time"` will be written as follows by `F_MifString`:

```
<String `space'>
<Char HardHyphen>
<String `time'>
```

## Support for legacy clients

FDK 8 provides a *Compatibility Mode* to support clients that were written for earlier releases of FrameMaker. When running in this mode, FDK APIs and FDE functions do not accept UTF-8 input and mimic the behavior of FDK 7.2 as closely as possible. APIs and FDE functions that existed in FDK 7.2 exhibit such behavior for any new properties as well. However, new APIs and functions added in FDK 8 exhibit no special behavior in *Compatibility Mode*.

### Enable *Compatibility Mode* in the FDK

*Compatibility Mode* is the default mode of operation for both the FDE and APIs and doesn't need to be enabled. If *Unicode Mode* has not been enabled for the FDE or APIs, they are running in *Compatibility Mode*.

This allows clients written for earlier releases of FrameMaker and compiled with FDK 7.2 or earlier to function correctly with FrameMaker 8 without being recompiled. This also minimizes the changes that clients need if they are recompiled with FDK 8.

The APIs and FDE functions provide backward compatibility to a large extent in *Compatibility Mode,* but there are certain limitations as mentioned in the following sections. FDK 8 is slower in *Compatibility Mode* than in *Unicode Mode*. The slowness and limitations of *Compatibility Mode* make *Unicode Mode* the recommended mode of operation. *Compatibility Mode* should be used, as far as possible, only to support clients that have not been modified to handle Unicode data.

You can explicitly set the APIs to work in *Compatibility Mode* if they were running in *Unicode Mode* by making the following call:

```
F_ApiEnableUnicode(False);
```

You can explicitly set the FDE to work in *Compatibility Mode* if it was running in *Unicode Mode* by setting the FDE encoding to any encoding other than `"UTF-8"` by calling

`F_FdeInitFontEncs`. For example, the following code sets the FDE encoding to `"FrameRoman"`, enabling *Compatibility Mode* for the FDE:

```
F_FdeInitFontEncs((ConStringT) "FrameRoman");
```

Another example of setting *Compatibility Mode* for the FDE is as follows:

```
StringT encName = F_ApiGetString(0,
                        FV_SessionId, FP_DialogEncodingName);
F_FdeInitFontEncs((ConStringT) encName);
```

The above code sets the encoding of the FDE to the Dialog Encoding of FrameMaker, which is dependent on the Operating System locale and has only five possible values that do not include `"UTF-8"` (for more details, refer to the Dialog Encoding section later in this document and the *FDK Programmer's Reference*). Running FDE functions or APIs in different modes of operations at different times is called *Mixed Mode* operation and isn't recommended.

## Behavior of FDK APIs in *Compatibility Mode*

In *Compatibility Mode*, FDK APIs expect all strings to be in the Dialog Encoding of FrameMaker. The Dialog Encoding of FrameMaker is dependent upon the OS locale settings and has five possible values: `FrameRoman, JISX0208.ShiftJIS, BIG5, GB2312-80.EUC,` and `KSC5601-1992.` To find the Dialog Encoding of FrameMaker, make the following call:

```
StringT dialogEnc = F_ApiGetString(0, FV_SessionId,
                                        FP_DialogEncodingName);
```

For locales like `en_US` and `en_US.UTF-8` on Solaris, the Dialog Encoding is FrameRoman. Therefore, when running in *Compatibility Mode* in these locales, FDK APIs expect all input arguments to have strings valid in the `FrameRoman` encoding and all return values to have strings in `FrameRoman` encoding. On a Japanese locale like `ja`, the same APIs transact in Shift-JIS encoding.

### Internal representation of strings in FrameMaker

In FrameMaker 7.2 and earlier releases, string inputs to APIs were considered as a sequence of bytes without any validation of encoding. You could, for example, set the name of a paragraph format in Shift-JIS encoding while running FrameMaker in an English locale (Dialog Encoding in an English locale is FrameRoman). The name of such a paragraph format, however, showed as garbled text in the English locale. The same document when opened in a Japanese locale (with Shift-JIS as the Dialog Encoding), showed the paragraph format name correctly, because the sequence of bytes set as the paragraph format name were interpreted as Shift-JIS.

In FrameMaker 8, the internal representation of all strings is UTF-8. Hence, the name of a paragraph format, once set in an 8 document (FM/MIF), is displayed in the same manner across all locales.

### When APIs are used to obtain strings that aren't representable in the Dialog Encoding of FrameMaker

FrameMaker 8 has Unicode support and can have characters from scripts like Arabic, Devanagiri, and others. When queried using an API in *Compatibility Mode*, strings containing such characters always contain question marks `'?'` instead of the characters because these aren't valid in any of the five Dialog Encodings.

Japanese strings are also stored internally as UTF-8 but have valid representations in Shift-JIS. Therefore, on a Japanese locale, when queried using an API in *Compatibility Mode*, the Shift-JIS equivalent of such strings are returned correctly. However, on an English locale, when queried in the same manner, the strings contain question marks `'?'` in place of any character that isn't representable in FrameRoman. No Japanese character, for example, is representable in FrameRoman. However, because Shift-JIS also contains the basic ASCII range, which is representable in FrameRoman, characters are converted correctly.

### Passing a string not in the Dialog Encoding of FrameMaker

If FDK APIs are running in *Compatibility Mode* on an English locale and you provide the input to an API in Shift-JIS encoding, this is incorrect because the API interprets the input as FrameRoman encoding. Even if the string is valid in the current Dialog Encoding, its interpretation in terms of glyphs is incorrect. For example, the Shift-JIS string " あぶい " has the byte representation `0x82 0xA0 0x82 0xD4 0x82 0xA2`, which is a valid byte sequence in FrameRoman encoding, but is actually interpreted as the string `"Ç†Ç‘Ç¢"` in FrameRoman encoding. When this input is passed to an FDK API in *Compatibility Mode* on an English locale, the API internally converts it to UTF-8 assuming it to be FrameRoman. If the API is used to set the paragraph format name and the file generated is subsequently stored as an 8 document (FM/MIF) and then opened on ANY locale, the paragraph format name displays as `"Ç†Ç‘Ç¢"` (and not as " あぶい " even on a Japanese locale).

If an API is provided a string in *Compatibility Mode*, a part or whole of which is invalid in the Dialog Encoding of FrameMaker, it is converted to a series of question marks `'?'`.

### Example of an API in Compatibility Mode

`F_ApiAlert` accepts Shift-JIS input in *Compatibility Mode* if Dialog Encoding is Shift-JIS.

```
F_ApiAlert("This is Shift-JIS: \x82\xA0\x82\xD4\x82\xA2",
     FF_ALERT_CONTINUE_NOTE);
```

This call produces the following alert on a Japanese locale (where Dialog Encoding is Shift-JIS):


The same call produces the following alert on an English locale (where Dialog Encoding is FrameRoman). This is because the byte sequence `0x82 0xA0 0x82 0xD4 0x82 0xA2` stands for the string `"Ç†Ç‘Ç¢"` in FrameRoman encoding.


### F_ApiNotify

In *Compatibility Mode*, this API behaves differently on Windows than it does on Solaris. On Windows, the string parameter `sparm` is in the Dialog Encoding. However on Solaris, the string parameter `sparm` is provided in the Locale encoding and not the Dialog Encoding (unlike most other APIs). This is because `sparm` is mostly used to pass filepaths, which must be in the Locale encoding on Solaris. For more information on filepath handling in *Compatibility Mode*, see the section on FDE FilePath functions.

### F_ApiGetSupportedEncodings

In *Compatibility Mode*, the list returned by the API `F_ApiGetSupportedEncodings` doesn't contain `"UTF-8"` because this encoding isn't supported in the *Compatibility Mode* in any locale. The behavior of this API in *Compatibility Mode* is the same as in FDK 7.2

### F_ApiIsEncodingSupported

In *Compatibility Mode*, `F_ApiIsEncodingSupported` returns `False` for `"UTF-8"`. The behavior of this API in *Compatibility Mode* is the same as in FDK 7.2

### F_ApiSave, F_ApiOpen, F_ApiImport

In *Compatibility Mode*, these APIs behave differently on Windows than they do on Solaris. On Windows, the filename is expected to be in the Dialog Encoding. However on Solaris, filepaths must be in the Locale encoding. For more information on filepath handling in *Compatibility Mode*, see the section on FDE FilePath functions.

In the FDK 8 *Compatibility Mode*, `FV_SaveFmtBinary` and `FV_SaveFmtInterchange` are equivalent to `FV_SaveFmtBinary80` and `FV_SaveFmtInterchange70` respectively when calling `F_ApiSave`. Therefore, the default mode of saving MIF in *Compatibility Mode* is legacy format (MIF7 and MIF Book 7). The default mode for saving FM files is 8 format (FM8 and FM Book 8).

### F_ApiAddText F_ApiGetText, F_ApiGetText2, F_ApiGetTextForRange, F_ApiGetTextForRange2

In *Compatibility Mode*, these APIs do not expect/return strings in the Dialog Encoding of FrameMaker. Instead they expect/return strings in the encoding of the font applied on the text.

Therefore if the font `MS PMincho` (encoding of Shift-JIS) is applied to the text " あぶい ", `F_ApiGetText` returns the byte sequence `0x82 0xA0 0x82 0xD4 0x82 0xA2`, which is the representation of " あぶい " in Shift-JIS. The same document can have the text `"ÆÐ"` with the font `Times New Roman` (encoding of FrameRoman) applied. For this text, `F_ApiGetText` returns `0xAE 0xC3`, which is the FrameRoman encoding of `"ÆÐ"`. Characters that aren't representable in the encoding of the applied fonts are converted to question marks. This behavior is common to all variants of `F_ApiGetText`.

`F_ApiAddText` has a similar behavior. It expects the input to be in the encoding of the font applied at the insertion point. Thus, if `MS PMincho` is applied at the insertion point, `F_ApiAddText` expects the input to be in Shift-JIS encoding. The font must be applied to a location before the addition of text so that `F_ApiAddText` can ascertain the encoding. If all the document fonts are in the same encoding as the text (including the default paragraph font), this will pose no problems. However if a document is likely to have mixed contents, for example if it has some Japanese content "あぶい" in `MS PMincho` (byte representation `0x82 0xA0 0x82 0xD4 0x82 0xA2`) right after some English content in `Times New Roman`, the font `MS PMincho` must be applied to the insertion location before adding the text. Otherwise, it is interpreted as being in FrameRoman encoding and is added to the document as the text `"Ç†Ç‘Ç¢"`.

The conversion from UTF-8 to single byte and vice-versa that must be performed for Symbol/Dingbats/Webdings in *Unicode Mode* is already handled by the API in *Compatibility Mode* (see the information for these same APIs in the *Unicode Mode* section). Therefore, getting the text ∇ (the Ñ with Symbol font applied) returns `0xD1`, and not `0xC3 0x91`, in *Compatibility Mode*. Similarly, for adding this character (when the font is Symbol), `F_ApiAddText` expects `0xD1`, and not `0xC3 0x91`.

### F_ApiGetEncodingForFont, F_ApiGetEncodingForFamily

The APIs `F_ApiGetEncodingForFont` and `F_ApiGetEncodingForFamily` have the same behavior as in FDK 7.2 in both *Compatibility Mode* and *Unicode Mode*. That is, these do not return `"UTF-8"` as the encoding for any font in either mode.

## Behavior of FDE functions in *Compatibility Mode*

In the FDE, functions found under the following headings in the *FDK Programmer's Reference* do not depend on the FDE encoding and, therefore, have the same behavior in *Unicode Mode* and *Compatibility Mode*:

- Characters
- F-Codes
- Fonts
- Hash Tables
- Memory: manipulating with handles
- Memory: manipulating with pointers
- Metrics
- String lists
- Strings: allocating, copying, and deallocating
- Strings: comparing and parsing
- Strings: concatenating
- Strings: miscellaneous
- Strings: encoded

These include functions that either do not deal with strings (for example `F_MetricSqrt`), expect strings/characters to be in FrameRoman encoding (for example, `F_StrReverse`), can work for any encoding other than UTF-16 or UTF-32 (for example, `F_StrCopyString`), or explicitly ask for the encoding of the strings (for example, `F_StrLenEnc`).

FDE functions found under the following headings in *FDK Programmer's Reference* depend on the FDE encoding and have different behaviors in *Unicode Mode* and *Compatibility Mode*. When *Compatibility Mode* is enabled for the FDE, these expect all strings in input parameters to be valid in the FDE encoding set by `F_FdeInitFontEncs`. Strings in all return values or parameters set by these functions in this mode will be in the FDE encoding.

### Debugging

The following API functions found in the *FDK Programmer's Reference* under the heading *Debugging* are a notable exception to conventions. These behave as a part of the FDE, rather than as a part of the APIs, and thus depend upon the FDE encoding. Their behavior is similar to the behavior of `F_Printf` (see under the I/O section).

- `F_ApiPrintTextItem`
- `F_ApiPrintTextItems`
- `F_ApiPrintPropVal`
- `F_ApiPrintPropVals`

#### Files, directories, and filepaths

In *Compatibility Mode*, all inputs are expected to be in FDE encoding and all outputs are also in this encoding. The system locale must also be compatible with the FDE encoding. For example, if the FDE encoding is `"JISX0208.ShiftJIS"`, then the locale must be Japanese on Windows and `ja` on Solaris for Japanese filenames to be used correctly.

In Windows, filepaths that contain invalid characters in the FDE encoding are inaccessible. The behavior of these functions on Solaris is slightly different from that on Windows. Solaris stores filenames in a byte-based manner, rather than in a specific encoding. For some filepath functions in *Compatibility Mode,* you might be able to deal with filepaths irrespective of their encodings. However, for full filepath function support, use only filepaths in the FDE encoding.

#### I/O

The calls `F_ChannelOpen` and `F_ChannelMakeTmp` handle filepaths in the same manner as filepath functions discussed above.

All data written to the NULL channel using `F_Printf`, `F_Warning` or `F_ChannelWrite` goes to the console. The behavior of these in *Compatibility Mode* is different in Windows and Solaris. On Windows, because the console window expects Unicode irrespective of the locale, data sent to the console is converted from the FDE encoding to UTF-8 before being displayed. Hence, these calls expect the data to be in the FDE encoding when writing to the console.

On Solaris, because the console window expects byte sequences and can only display byte sequences valid on the current locale, the data sent for printing on the console must be in the locale encoding.

#### Maker Interchange Format (MIF)

In *Compatibility Mode*, these functions behave as they did in FDK 7.2. Note that only F_MifText is capable of dealing with double-byte strings, and the rest expect FrameRoman strings. None of these functions depend on the FDE encoding in *Compatibility Mode*.

## Structured Import/Export APIs

The FDK 8, Structured APIs do not provide a *Compatibility Mode* of operation. The behavior of these APIs is dependent on two aspects: the statically linked code that resides in `struct.lib` shipped with the FDK, and the FDK APIs that this code internally calls.

The FDK APIs provide a *Compatibility Mode,* which is the default mode of operation until *Unicode Mode* is enabled by making the `F_ApiEnableUnicode(True)`. However, the static code in `struct.lib` in FDK 8 doesn't have a *Compatibility Mode,* and these APIs might fail or produce unpredictable results with non-Unicode data.

If you want legacy behavior (*Compatibility Mode*) in structured clients, therefore, you must use the `struct.lib` shipped with FDK 7.2 or earlier. In addition, you must run FDK APIs in

*Compatibility Mode*. Legacy clients that have not been recompiled with FDK 8 run in *Compatibility Mode* without a problem.

For 8 behavior (*Unicode Mode*), you must link the client against `struct.lib` shipped with FDK 8 and run FDK APIs in *Unicode Mode*.

## International Components for Unicode (ICU)

In FDK 8, FDE functions rely extensively on ICU. This has some important consequences for both legacy and FDK 8 clients.

### Set the ICU data directory

For correct functionality, ICU requires convertor data. Any process that uses ICU must initialize ICU by setting its data directory. FrameMaker 8 also uses ICU internally and initializes it by setting the data directory to `fminit/icu_data` where all ICU convertor data shipped with FrameMaker is stored. Because the FDK 8 FDE uses ICU extensively, all clients compiled with FDK 8 must also set the ICU data directory correctly by making the following call (where `icu_data_dir` is the directory that stores the ICU convertor data). The ICU data directory must be in ASCII only. A network pathname (UNC) can be used as the data directory.

```
F_SetICUDataDir(icu_data_dir);
```

Setting the ICU data directory has a process-wide effect. For example, because synchronous DLL clients on Windows reside in the same process as FrameMaker, which also initializes ICU for internal usage, such clients do not need to set the ICU data directory explicitly. Setting the ICU data directory incorrectly can adversely affect FrameMaker if the client is in the same process space. Use the call `F_GetICUDataDir` to query the current ICU data directory being used within a process.

Clients that do not reside in the same process space as FrameMaker must set the ICU data directory for correct functionality. Wherever possible, `F_FdeInit` attempts to set the ICU data directory if it has not already been set. Because `F_FdeInit` attempts to pick up the ICU data directory path information from the instance of FrameMaker that the client is connected to, it can set ICU data directory properly only when the client is connected to a FrameMaker session running on (and from) the same machine at the time of the call.

Clients should attempt to set the ICU data directory themselves. This is particularly important for remote clients and asynchronous clients that sometimes make FDE calls without being attached to a FrameMaker session.

### F_FdeInit as the first FDE call

Using FDK 7.2, you could sometimes write clients that did not call `F_FdeInit` or that called other FDE functions first.

However, in FDK 8, the dependence on internal FDE structures being initialized is greater, and `F_FdeInit` attempts to initialize the ICU data directory for the client if it has not already been set. Therefore, you must make the `F-Fdelnit` call before making any other FDE call in FDK 8. Exceptions are `F_SetICUDataDir` and `F_GetICUDataDir`, which you can call before calling this function.

### Dependency on ICU .so files at compile time and run time (Solaris only)

On Solaris, all clients compiled with FDK 8 also need to link to ICU .so files, and these files must be in the system search path at run time. You can download the .so files from the ICU website, or find them in the `FMHome/bin/sunxm.s5.sparc` folder. FrameMaker 8 uses the 2.8 version of ICU. The following libraries must be linked against:

```
libicudata.so
libicuuc.so
libicuio.so
libicui18n.so
libicutoolutil.so
libicule.so
```

Modify the link flags to include the following (where `LICUDIR` is the location of .so files):

```
-L$(LICUDIR) -licudata -licuuc -licuio -licui18n -licutoolutil -licule
```

You must also modify the environment variable `LD_LIBRARY_PATH` to include the directory containing the ICU .so files.

### Dependency on ICU DLL files at run time (Windows only)

All FDK 8 clients need ICU DLLs at run time. For synchronous clients, FrameMaker 8 takes care of loading these DLLs. Asynchronous clients must make sure that the required ICU DLLs are present in any of the following locations:

- A directory which is in the system search path
- Directory where the client's executable resides
- Directory from which the client is executed

ICU DLL files are shipped with the FrameMaker 8 Windows version and can be found in the `FMHome` folder. These are also available for download from ICU website.

## Legacy clients that have not been re-compiled with FDK 8

Most legacy clients that have been compiled with an earlier release of the FDK will work with FrameMaker 8 without modification. This section summarizes the behavior of FDK for such clients. *For more details, refer the sections 'Support for legacy clients' on page 11 and 'Structured Import/Export APIs' on page 17*

## Behavior of FDK APIs

Clients compiled with FDK 7.2 or earlier won't have made the `F_ApiEnableUnicode(True)` call that has been added in FDK 8. Therefore, all APIs will run in the FDK 8 *Compatibility Mode*, which provides backward compatibility for legacy clients. The FDK APIs in this mode closely imitate 7.2 versions of the APIs.

### All string inputs must be in the FrameMaker Dialog Encoding

A client that calls FDK APIs with strings in an encoding other than the Dialog Encoding of FrameMaker won't work correctly. Although this scenario was possible in FrameMaker 7.2 and earlier, the strings passed through such APIs appeared garbled. When the document was opened in a locale where the Dialog Encoding of FrameMaker was the same as the encoding of the strings, the previously garbled text appeared correctly.

In version 8, this scenario is no longer supported. FDK APIs must only be called with strings in the Dialog Encoding. The Dialog Encoding can be changed by changing the locale. The following section explains exceptions to this rule.

### F_ApiAddText, F_ApiGetText, F_ApiGetText2, F_ApiGetTextForRange, F_ApiGetTextForRange2

These APIs do not expect/return strings in the Dialog Encoding of FrameMaker. Instead, they expect/return strings in the encoding of the font applied on the text.

### F_ApiSave

For clients running FDK APIs in *Compatibility Mode*, `F_ApiSave` saves documents/books in FM 8 and MIF 7.0 if provided `FV_SaveFmtBinary` and `FV_SaveFmtInterchange` respectively. To make these clients save documents/books in MIF 8 format, you must recompile and either enable *Unicode Mode* or provide the new parameter value `FV_SaveFmtInterchange80` while calling `F_ApiSave`.

## Behavior of FDE functions

The behavior of version 7.2 or earlier FDE functions isn't affected by version 8 in clients that maintain their own copy of the FDK.

The only clients that do not maintain their own copy of FDK are Dynamically Linked (DL) clients, which are possible only on Solaris. Because these clients share FrameMaker's copy of the FDK, they have access to the 8 version of the FDE. However, because FrameMaker already sets the ICU data directory for the shared copy of the FDE and calls `F_FdeInit`, these clients do not need to make any modification. The FDE encoding is set to FrameRoman by default, so they run in the FDK 8 *Compatibility Mode*.

## Behavior of Structured Import/Export APIs

Because legacy clients that have not been recompiled with FDK 8 maintain their own copy of `struct.lib`, Structured APIs are affected only by FDK API calls made inside `struct.lib`. However, because FDK APIs run in *Compatibility Mode* by default, Structured APIs provide the same level of backward support as FDK APIs.

## Recompiling old clients with FDK 8

Certain clients written for FrameMaker 7.2 and earlier might not function correctly if recompiled with FDK 8. Because FDK 8 depends upon ICU, compiling with FDK 8 introduces an ICU dependency in the client. On Solaris, the linking must be modified to include ICU libraries (ICU .so files). The code might have to be modified to set the ICU data directory correctly.

Also, for asynchronous clients there is a run-time dependency on ICU DLLs on Windows and on ICU .so files on Solaris. On Windows, the ICU DLLs must be present in any one of the following locations (a) a directory which is in the system search path (b) in the directory where the client's executable resides or (c) in the directory from which the client is executed. On Solaris, the .so files must be present in a path that is in the `LD_LIBRARY_PATH` environment variable.

Clients that make calls to other FDE functions before calling `F_FdeInit` might not work correctly. All clients must ensure that they call `F_FdeInit` before any other FDE call (except `F_SetICUDataDir` and `F_GetICUDataDir`).

Recompiling with FDK 8 means that the 8 version of FDE functions become available to the client. The FDE encoding, unless set explicitly, is set to FrameRoman. Therefore, the FDE runs in *Compatibility Mode* by default. Unless the code of the client has been modified to enable *Unicode Mode* for APIs, the APIs also run in *Compatibility Mode*.

Recompiling structured clients with FDK 8 implies use of the new `struct.lib`. This library expects Unicode data. For correct *Compatibility Mode* functionality, the client must be compiled against the `struct.lib` provided in earlier releases of the FDK.

*For more details, refer the sections 'Unicode Support in FrameMaker' on page 5, 'Support for legacy clients' on page 11, 'Structured Import/Export APIs' on page 17 and 'International Components for Unicode (ICU)' on page 18*

## Asynchronous clients

All asynchronous clients compiled with FDK 8 must set the ICU data directory correctly by making the `F_SetICUDataDir` call. They must also link against ICU .so files (if on Solaris) and must make sure that ICU DLLs or .so files are present in the appropriate search paths. For details, see the previous sections that describe the dependencies on these files.

### FrameMaker Product Activation

Asynchronous clients on Windows that launch a FrameMaker process and wait for it to become idle (by calling `WaitForInputIdle`) before attempting to connect using `F_ApiWinConnectSession` might encounter some problems.

If FrameMaker has not been activated (relevant only for the FrameMaker Point Product on Windows XP and Windows Vista®), an activation screen prompts the user for input. The

`WaitForInputIdle` call returns at this point while FrameMaker isn't actually ready for communication. Therefore, the client must give the user enough time to activate or skip activation before attempting to connect to FrameMaker using `F_ApiWinConnectSession`. Otherwise, the client can require the user to activate the product before using it.

Despite activation, `WaitForInputIdle` returns too early before FrameMaker is actually ready to establish a connection. The issue can be resolved by modifying the code and introducing a 5-10 second sleep before attempting to connect to the FrameMaker session. Another solution is to attempt to connect a multiple or indefinite number of times with short sleeps in between.

Asynchronous clients running the FrameMaker Point Product on Windows 2000 or running the FrameMaker Server on Windows 2000, XP, or VISTA won't encounter any such problems and do not need modification.

## Working with Microsoft Visual C++ 2003 & Visual C++ 6.0

Because the new FDK 8 uses Microsoft Development Environment 2005 (Version 8), FDK clients should be compiled in this same environment. If you want to use Microsoft Visual C++ 2003 (Version 7.1), you must add the following lines to the source code of the FDK clients:

```
extern "C" {
long _ftol2(double);
long _ftol2_sse(double dblSource) { return _ftol2(dblSource); } }
```

If you want to use Microsoft Visual C++® 6.0, you must add the following lines to the source code of the FDK clients:

```
extern "C" {
long _ftol(double);
long _ftol2_sse(double dblSource) { return _ftol(dblSource); } }
```

In addition, you must link against the library `bufferoverflowU.lib` to solve `___security_cookie` and `___security_cookie@4` linker errors. These linker errors happen because FDK 8 was compiled with VS 2005 with the `/GS` switch. This switch puts in buffer overflow protection, which was unavailable in VC 6.0. The library `bufferoverflowU.lib` can be found in the Platform SDK for Windows Server® 2003, which is downloadable from `microsoft.com`. It can also be found in the VS 2005 folder in `VC\PlatformSDK\Lib`.

**IMPORTANT:** *Although these fixes resolve linker issues, the behavior under these fixes is unpredictable, and we strongly recommend using Visual C++ 2005.*

## Modified APIs

The following APIs exhibit unique behaviors in *Unicode Mode* and *Compatibility Mode*. This has been documented in the respective sections.

- `F_ApiGetSupportedEncodings`
- `F_ApiIsEncodingSupported`
- `F_ApiSave`
- `F_ApiAddText`
- `F_ApiGetText`
- `F_ApiGetText2`
- `F_ApiGetTextForRange`
- `F_ApiGetTextForRange2`

### F_ApiGetImportDefaultParams

- `FS_NumCellSeparators` now has a default value of 1 instead of 0.
- `FS_InsetData` property has been added and has a default value of 0.
- `FS_UseHTTP` property has been added and has a default value of 0.

### F_ApiInitialize

The behavior of the callback `F_ApiInitialize` has changed. It is now called for all API clients, including filters and document reports. This enables filters and document reports to make the `F_ApiEnableUnicode(True)` call necessary for enabling *Unicode Mode*. If *Unicode Mode* isn't enabled, `sparm` received in `F_ApiNotify` isn't in Unicode because the API runs in *Compatibility Mode*.

# New APIs

## F_ApiEnableUnicode()

Used to enable *Unicode Mode* or *Compatibility Mode*

### Synopsis

```
#include "fapi.h"
...
VoidT F_ApiEnableUnicode(BoolT enable);
```

### Arguments

| | |
|---|---|
| `enable` | `True` enables *Unicode Mode*, `False` enables *Compatibility Mode.* |

### Returns

`VoidT`

### Example

The following example creates two alerts. The first one shows the string `"This will be treated as FrameRoman on English locale: Ð§ ÎÆ§ ÿ¥"` on an English locale. The second one shows the string "`This will be treated as Unicode on every locale:` ä 뮤 ش " on any locale.

```
#include "fencode.h"
. . .
F_ApiAlert("This will be treated as FrameRoman on English locale:
\xC3\xA4 \xEB\xAE\xA4 \xD8\xB4",
     FF_ALERT_CONTINUE_NOTE);

F_ApiEnableUnicode(True);
F_ApiAlert("This will be treated as Unicode on every locale: \xC3\xA4
\xEB\xAE\xA4 \xD8\xB4",
     FF_ALERT_CONTINUE_NOTE);
. . .
```

# New and changed FDK objects, properties, property value constants and script parameter values

The following sections describe new or modified properties and property values for existing objects. Properties and values that have not changed from earlier releases aren't listed here.

## New Objects

FO_AttrCondExpr

## New Properties:

### Session Properties

FP_TrackChangesOn

### FO_Inset Properties

FP_INSETinfo

FP_InsetURL

FP_NoFlashInPDF

FP_No3DInPDF

### FO_AttrCondExpr Properties

FP_FirstAttrCondExprInDoc

FP_NextAttrCondExprInDoc

FP_PreviewState

FP_AttrCondExprStr

FP_AttrCondExprIsActive

FP_BooleanConditionExpression

## New Property Value Constants

### FP_PreviewState Values

FV_PREVIEW_OFF_TRACK_CHANGE

FV_PREVIEW_ON_ORIGINAL

FV_PREVIEW_ON_FINAL

#### FS_FileType Values

FV_SaveFmtBinary70

FV_SaveFmtBinary80

FV_SaveFmtInterchange70

FV_SaveFmtInterchange80

### Deprecated Property Value Constants

#### FS_FileType Values

FV_SaveFmtBinary60

FV_SaveFmtInterchange60

FV_SaveFmtStationary

### New Script Parameter

FS_InsetData

FS_UseHTTP

### FP_DialogEncodingName

The possible values and the behavior of FrameMaker Dialog Encoding remains the same. This behavior is the same in both *Compatibility Mode* and *Unicode Mode*. Obtaining the value of FP_DialogEncodingName by making the call F_ApiGetString(0, FV_SessionId, FP_DialogEncodingName) won't return "UTF-8" even when running FrameMaker on a UTF-8 locale. Therefore, clients that used to initialize the FDE using the Dialog Encoding must explicitly set the FDE encoding to UTF-8 if they so desire.

## New Notifications

FA_Note_U3DCommand

FA_Note_Not_U3DCommand

FA_Note_RSC_Supported_File

FA_Note_Not_RSC_Supported_File

## New File Types Available for Import

The following table shows the hint strings that can be used with `FP_ImportHint` to import additional file types added in FrameMaker 8. Note that the filter name is a four-byte code. If the name string contains fewer than four alphanumeric characters, the remaining bytes are filled out with spaces.

| File type | Hint string |
|---|---|
| U3D | `"0001ADBIU3D "` |
| SWF | `"0001ADBISWF "` |
| Unicode Text Files | |
| UTF-8 | `"0001PGRFTUT8"` |
| UTF-16LE | `"0001PGRFTU1L"` |
| UTF-16BE | `"0001PGRFTU1B"` |
| UTF-32LE | `"0001PGRFTU3L"` |
| UTF-32BE | `"0001PGRFTU3B"` |

## Enhancement of the RTF filter

The RTF 1.6 filter has been enhanced for Unicode support. The RTF 1.3 filter has been deprecated because it is unable to handle Unicode correctly and might crash. It should not be used.

## New FDE Types

`UChar32T` is 4 bytes and should be used to store a UTF-32 code unit.

`UChar16T` is 2 bytes and should be used to store a UTF-16 code unit.

## String handling functions in FDE

### Handling of Unicode Characters

Many newly added calls like `F_StrChrUTF8` and `F_CharIsLowerUTF8` accept a Unicode character in UTF-8 encoding. The character can be passed to these functions in the form of a `UCharT` pointer, a `StringT`, or a `ConStringT`. In the following example, the function `F_CharUTF8ToUTF32` treats the sequence `0xE2 0x80 0x93` as a single character EM DASH '—' in UTF-8:

```
#include "fencode.h"
...
UChar32T emDash_UTF32;
UCharT emDash[3];
emDash[0]=0xE2;
emDash[1]=0x80;
emDash[2]=0x93;
emDash_UTF32 = F_CharUTF8ToUTF32((ConStringT) emDash);
```

The functions expecting Unicode characters in this manner only parse the `UCharT` sequence enough to pick up one character. Therefore, the `UCharT` sequence need not be terminated by a null byte when being passed as a character. If a `UCharT` sequence contains more than one UTF-8 character, only the first character is considered. You must provide valid sequences containing at least one Unicode character to these functions.

### Truncation of Unicode Strings

All FDE functions, unless stated otherwise, expect and return lengths of UTF-8 strings in terms of bytes, rather than the number of Unicode characters. Some FDE string handling functions either restrict a string to a certain number of bytes, or consider the strings only up to a certain number of bytes for performing comparisons or other operations. An example of this is the `StrCpyN` function, which copies at most N bytes (including the terminating null byte) from one string into another.

Some of these functions might truncate a UTF-8 string at an invalid boundary. In the following example, `F_StrTrunc` truncates the UTF-8 string "A—B" at an invalid boundary, rendering it

invalid. The effect of this call is to truncate the string to `"\x41\xE2\x80"` (midway between the EM DASH character '—').

```
StringT s = F_StrCopyString((ConStringT) "\x41\xE2\x80\x93\x42");
F_StrTrunc(s, 3);
```

Such functions aren't safe for UTF-8 input. Certain functions of this type that have been labeled UTF-8 safe do not truncate strings at an invalid boundary. For example, `F_StrTruncEnc` is UTF-8 safe when called with UTF-8 as the encoding.

```
StringT s = F_StrCopyString((ConStringT) "\x41\xE2\x80\x93\x42");
FontEncIdT feId = F_FontEncId((ConStringT) "UTF-8");
F_StrTruncEnc(s, 3, feId);
```

The above call truncates the string to `"\x41"` (or "A"), which is the last complete UTF-8 character in the invalid string `"\x41\xE2\x80"`. In a similar manner, in the following call, `F_StrCmpNEnc` returns `True`, while `F_StrCmpN` doesn't for the same strings "ÆÐ" and "ÆØ", and the same length 3.

```
StringT s1 = F_StrCopyString((ConStringT) "\xE2\x80\x93");
StringT s2 = F_StrCopyString((ConStringT) "\xE2\x80\x93");
FontEncIdT feId = F_FontEncId((ConStringT) "UTF-8");

if (F_StrCmpNEnc(s1,s2,3,feId)==0)
     F_Printf(NULL,(ConStringT) "\nF_StrCmpNEnc:%s and %s are equal
                  on %d bytes", s1, s2, 3);
else
     F_Printf(NULL,(ConStringT) "\nF_StrCmpNEnc:%s and %s are not equal
                  on %d bytes", s1, s2, 3);

if (F_StrCmpN(s1,s2,3)==0)
     F_Printf(NULL,(ConStringT) "\nF_StrCmpN:%s and %s are equal on
                  %d bytes", s1, s2, 3);
else
     F_Printf(NULL,(ConStringT) "\nF_StrCmpN:%s and %s are not equal
                  on %d bytes", s1, s2, 3);
```

The code produces the following output on the console window:

```
F_StrCmpNEnc:ÆÐ and ÆØ are equal on 3 bytes
F_StrCmpN:ÆÐ and ÆØ are not equal on 3 bytes
```

## Special remarks on UTF-16 and UTF-32

No FDK API can handle UTF-16 or UTF-32 input. FDE functions that can handle UTF-16 and UTF-32 input expect the input to be in the endianness of the Operating System unless explicitly stated otherwise. Therefore, on Windows, these functions expect the input to be in UTF-16LE and UTF-32LE, while on Solaris, UTF-16BE and UTF-32BE are expected.

Because a code unit in UTF-16 and UTF-32 encodings is 2 and 4 bytes respectively, a string in these encodings needs the sequence `0x00 0x00` and `0x00 0x00 0x00 0x00` respectively to indicate string termination. Single `0x00` bytes might occur multiple times in a string without being interpreted as a terminating character. For example, the string "AB" is represented as `0x41 0x00 0x42 0x00` in UTF-16LE. Therefore, many functions like `F_StrLen` that treat a single occurrence of `0x00` as an indication of string termination won't work correctly on UTF-16 and UTF-32 strings. Therefore, a function that has been explicitly indicated as UTF-16 or UTF-32 safe should not be passed strings in these encodings.

## Old string handling functions

The following FDE functions should not be used for UTF-8 strings as they can truncate a UTF-8 string midway through a character, rendering it invalid:

```
F_StrCpyN, F_StrCmpN, F_StrEqualN, F_StrICmpN, F_StrIEqualN,
F_StrPrefixN, F_StrCatN, F_StrCatIntN, F_StrCatCharN, F_StrTrunc
```

The following functions also do not handle Unicode properly because they work on a byte-wise basis and are UTF-8 unsafe:

```
F_StrTok, F_StrBrk, F_StrChr, F_StrRChr, F_StrReverse, F_StrStrip
```

The following functions also work on a byte-wise basis but are UTF-8 safe. They do not render UTF-8 strings invalid and can work for UTF-8 input to some extent. Because of the nature of UTF-8 encoding, `F_StrCmp` and `F_StrEqual` can be used to check Unicode code point-based equality. The functions `F_StrPrefix`, `F_StrSubString`, and `F_StrSuffix` also work correctly if given valid UTF-8 input. The case-insensitive versions of these functions are safe as well but can only take the case of the English alphabet (A-Z) into consideration.

```
F_StrCmp, F_StrEqual, F_StrICmp, F_StrIEqual, F_StrPrefix, F_StrIPrefix,
F_StrSubString, F_StrSuffix
```

The following functions can still be used for Unicode without any problems:

```
F_StrStripLeadingSpaces, F_StrStripTrailingSpaces, F_StrNew,
F_StrCopyString, F_StrCpy, F_Free, F_ApiDeallocateString, F_StrCat,
F_StrLen
```

The following functions can also be used for Unicode but won't deal with numbers written in different scripts (like Hindi and Arabic) properly:

```
F_StrAlphaToInt, F_StrAlphaToReal
```

## Modified string handling functions

FDE string handling functions that have an `Enc` suffix could handle double-byte encodings in FDK 7.2. In FDK 8, these can handle UTF-8 as well. The exceptions are:

```
F_StrChrEnc, F_StrRChrEnc, F_StrCatDblCharNEnc
```

The Enc functions that can accept UTF-8 never cut a Unicode string midway of a character's bytes and ensure that the string is valid if the input was valid. So, a function like F_StrCpyNEnc that copies at most N-1 bytes might copy fewer if cutting the string at the N-1 character makes it invalid. These functions cut strings at (or consider the string until) the first valid boundary before the point specified. These include:

```
F_StrTruncEnc, F_StrCatNEnc, F_StrNCatNEnc, F_StrCpyNEnc
and F_StrCmpNEnc, F_StrICmpNEnc, F_StrIEqualNEnc
```

The Enc functions compare Unicode characters by the code points (character-by-character, not byte-by-byte). Case-insensitive comparison is done by conversion to lower case followed by comparison on code points. Because of the UTF-8 encoding design, byte-by-byte comparison is equivalent to code-point-by-code-point comparison. The functions that perform comparisons are:

```
F_StrIEqualEnc, F_StrIEqualNEnc, F_StrICmpEnc, F_StrMCmpEnc,
F_StrCmpNEnc, F_StrICmpNEnc, F_StrQsortCmpEnc, F_StrIPrefixEnc,
F_StrISuffixEnc, F_StrStrEnc
```

The function F_StrLenEnc returns the number of Unicode characters in the string. Use F_StrLen to get the number of bytes.

## New string handling functions

### F_StrChrUTF8()

Returns a pointer to the first occurrence of a UTF-8 character in a UTF-8 string

**Synopsis**

```
#include "fencode.h"
. . .
StringT F_StrChrUTF8(ConStringT s, const UCharT *c);
```

**Arguments**

| | |
|---|---|
| s | The string to search |
| c | The character (in terms of a byte sequence) to search s for |

**Returns**

A pointer to the first occurrence of c in s, or NULL if c doesn't occur in s

**Example**

The following code prints "Бог и взял". Here the long hex sequence "\xD0\x91...\xD0\xBB" is the representation of "Бог дал, Бог и взял" in UTF-8 and "\xD0\xBB" of the character '?'.

```
#include "fencode.h"
. . .
StringT s, c, string;
F_FdeInitFontEncs((ConStringT)"UTF-8");

s =
F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xB3\x20\xD0\xB4\xD0\xB0\xD0\xBB\x2
C\x20\xD0\x91\xD0\xBE\xD0\xB3\x20\xD0\xB8\x20\xD0\xB2\xD0\xB7\xD1\x8F\xD
0\xBB");

c = "\xD0\xBB";

string = F_StrChrUTF8(s, c);
F_UTF8NextChar(&string); /* To skip the comma */
F_UTF8NextChar(&string); /* To skip the blank space */
F_Printf(NULL, "%s\n", string);
. . .
```

## F_StrRChrUTF8()

Returns a pointer to the first occurrence of a UTF-8 character in a UTF-8 string, starting from the end of the string

### Synopsis

```
#include "fencode.h"
. . .
StringT F_StrRChrUTF8(ConStringT s, const UCharT *c);
```

### Arguments

| | |
|---|---|
| s | The string to search |
| c | The character (in terms of a byte sequence) to search s for |

### Returns

A pointer to the first occurrence of c in s, or NULL if c doesn't occur in s

### Example

The following code prints "Бог и взял". Here the long hex sequence "\xD0\x91...\xD0\xBB" is the representation of "Бог дал, Бог и взял" in UTF-8 and "\xD0\x91" of the character 'Б'.

```
#include "fencode.h"
. . .
StringT s, c, string;
F_FdeInitFontEncs((ConStringT)"UTF-8");

s =
F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xB3\x20\xD0\xB4\xD0\xB0\xD0\xBB\x2
C\x20\xD0\x91\xD0\xBE\xD0\xB3\x20\xD0\xB8\x20\xD0\xB2\xD0\xB7\xD1\x8F\xD
0\xBB");

c = "\xD0\xBB";

string = F_StrRChrUTF8(s, c);
F_Printf(NULL, "%s\n", string);
. . .
```

## F_StrCmpUTF8()

Compares the Unicode code point of each character in two null-terminated UTF-8 strings

### Synopsis

```
#include "fencode.h"
. . .
IntT F_StrCmpUTF8(ConStringT s1, ConStringT s2);
```

### Arguments

| | |
|---|---|
| `s1` | A string |
| `s2` | A string to compare to `s1` |

### Returns

An integer greater than `0` if `s1` is lexicographically greater than `s2`; `0` if the strings are equal; or an integer less than `0` if `s1` is less than `s2`

### Example

The following code prints the string `The strings Бог and Бог are equal`

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
StringT s1 = F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xB3");
StringT s2 = F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xB3");
if(!F_StrCmp(s1, s2))
     F_Printf(NULL, "The strings %s and %s are equal\n", s1, s2);
. . .
```

The following code prints the string Бог дал is less than Бог и взял

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");

StringT s1 =
F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xB3\x20\xD0\xB4\xD0\xB0\xD0\xBB");

StringT s2 =
F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xB3\x20\xD0\xB8\x20\xD0\xB2\xD0\xB
7\xD1\x8F\xD0\xBB");

if(F_StrCmp(s1, s2)<0)
     F_Printf(NULL, "%s is less than %s\n", s1, s2);
else
     F_Printf(NULL, "%s is greater than %s\n", s1, s2);
. . .
```

## F_CharToLowerUTF8()

Converts a UTF-8 character to lowercase

### Synopsis

```
#include "fencode.h"
. . .
VoidT F_CharToLowerUTF8(const UCharT *c, UCharT *newchar);
```

### Arguments

| | |
|---|---|
| `c` | The character to convert |
| `newchar` | Pointer to the converted character. It must point to a block of modifiable memory large enough to store the converted character (a UTF-8 character requires at most 4 bytes). |

### Returns

`VoidT`

### Example

The following code prints `Lowercase of Δ is δ`

```
#include "fencode.h"
. . .
UCharT upper[]={0xCE, 0x94};
UCharT lower[4];
F_FdeInitFontEncs((ConStringT)"UTF-8");
F_CharToLowerUTF8(upper, lower);
F_Printf(NULL,"Lowercase of %C is %C", upper, lower);
...
```

## F_CharToUpperUTF8()

Converts a UTF-8 character to uppercase

### Synopsis

```
#include "fencode.h"
. . .
VoidT F_CharToUpperUTF8(const UCharT *c, UCharT *newchar);
```

### Arguments

| | |
|---|---|
| `c` | The character to convert |
| `newchar` | Pointer to the converted character. It must point to a block of modifiable memory large enough to store the converted character (a UTF-8 character requires at most 4 bytes). |

### Returns

`VoidT`

### Example

The following code prints `Uppercase of δ is Δ`

```
#include "fencode.h"
. . .
UCharT upper[4];
UCharT lower[] = {0xCE, 0xB4};
F_FdeInitFontEncs((ConStringT)"UTF-8");
F_CharToUpperUTF8(lower, upper);
F_Printf(NULL,"Uppercase of %C is %C", lower, upper);
...
```

### F_CharIsLowerUTF8()

Determines whether a specified UTF-8 character is a lowercase character

**Synopsis**

```
#include "fencode.h"
. . .
BoolT F_CharIsLowerUTF8 (const UCharT *c);
```

**Arguments**

| | |
|---|---|
| `c` | The character to check |

**Returns**

A nonzero value if the character is a lowercase character, or `0` if it isn't

**Example**

The following code prints δ `is in Lowercase`

```
#include "fencode.h"
. . .
UCharT lower[] = {0xCE, 0xB4};
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_CharIsLowerUTF8(lower))
     F_Printf(NULL,"%C is in Lowercase", lower);
...
```

### F_CharIsUpperUTF8()

Determines whether a specified UTF-8 character is an uppercase character

**Synopsis**

```
#include "fencode.h"
. . .
BoolT F_CharIsUpperUTF8 (const UCharT *c);
```

**Arguments**

| | |
|---|---|
| `c` | The character to check |

**Returns**

A nonzero value if the character is an uppercase character, or `0` if it isn't

**Example**

The following code prints Δ `is in Uppercase`

```
#include "fencode.h"
. . .
UCharT upper[]={0xCE, 0x94};
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_CharIsUpperUTF8(upper))
     F_Printf(NULL,"%C is in Uppercase", upper);
...
```

## F_CharIsAlphaUTF8()

Determines whether a specified UTF-8 character is an alphabetic character

**Synopsis**

```
#include "fencode.h"
. . .
BoolT F_CharIsAlphaUTF8 (const UCharT *c);
```

**Arguments**

| | |
|---|---|
| `c` | The character to check |

**Returns**

A nonzero value if the character is an alphabetic character, or `0` if it isn't

**Example**

The following code prints Δ `is Alphabetic`

```
#include "fencode.h"
. . .
UCharT upper_delta[]={0xCE, 0x94};
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_CharIsAlphaUTF8(upper_delta))
     F_Printf(NULL,"%C is Alphabetic", upper_delta);
...
```

## F_CharIsNumericUTF8()

Determines whether a specified UTF-8 character is a numeric character in a decimal system.

**NOTE:** This function doesn't return `True` for numeric characters like Ⅳ (code point 0x2163) that aren't in a decimal system.

**Synopsis**

```
#include "fencode.h"
. . .
BoolT F_CharIsNumericUTF8 (const UCharT *c);
```

**Arguments**

| | |
|---|---|
| `c` | The character to check |

**Returns**

A nonzero value if the character is a numeric character in a decimal system, or `0` if it isn't

**Example**

The following code prints `ৼ is Numeric`

```
#include "fencode.h"
. . .
StringT devanagiri_four="\xE0\xA5\xAA";
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_CharIsNumericUTF8(devanagiri_four))
     F_Printf(NULL,"%C is Numeric", devanagiri_four);
...
```

## F_CharIsAlphaNumericUTF8()

Determines whether a specified UTF-8 character is an alphanumeric character

**NOTE:** This function doesn't return `True` for numeric characters like Ⅳ (code point 0x2163) that aren't in a decimal system.

**Synopsis**

```
#include "fencode.h"
. . .
BoolT F_CharIsAlphaNumericUTF8 (const UCharT *c);
```

**Arguments**

| | |
|---|---|
| `c` | The character to check |

**Returns**

A nonzero value if the character is an alphanumeric character, or `0` if it isn't

**Example**

The following code prints Δ `is AlphaNumeric`

```
#include "fencode.h"
. . .
UCharT upper_delta[]={0xCE, 0x94};
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_CharIsAlphaNumericUTF8(upper_delta))
     F_Printf(NULL,"%C is AlphaNumeric", upper_delta);
...
```

## F_CharIsHexadecimalUTF8()

Determines whether a specified UTF-8 character is a hexadecimal digit (0, 1...9, a...f, or A...F)

### Synopsis

```
#include "fencode.h"
. . .
BoolT F_CharIsHexadecimalUTF8 (const UCharT *c);
```

### Arguments

| | |
|---|---|
| `c` | The character to check |

### Returns

A nonzero value if the character is an alphabetic character, or `0` if it isn't

### Example

The following code prints `F is Hexadecimal`

```
#include "fencode.h"
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_CharIsHexadecimalUTF8(&"F"))
     F_Printf(NULL,"F is Hexadecimal");
...
```

## F_CharIsEolUTF8()

Determines whether a specified character is a FrameMaker end-of-line (EOL) character. It returns `True` for decimal values `9` (forced return), `10` (end-of-paragraph), and `11` (end-of-flow).

**Synopsis**

```
#include "fencode.h"
. . .
BoolT F_CharIsEolUTF8 (const UCharT *c);
```

**Arguments**

| | |
|---|---|
| `c` | The character to check |

**Returns**

A nonzero value if the character is a FrameMaker end-of-line (EOL) character, or `0` if it isn't

**Example**

The following code prints `The character is an EOL character`

```
#include "fencode.h"
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_CharIsEolUTF8(&"\x0B"))/* decimal value of 0x0B is 11 */
     F_Printf(NULL,"The character is an EOL character);
...
```

## F_CharIsControlUTF8()

Determines whether a specified character is a FrameMaker control character. It returns `True` for decimal values `8` (tab), `9` (forced return), `10` (end-of-paragraph), and `11` (end-of-flow).

**Synopsis**

```
#include "fencode.h"
. . .
BoolT F_CharIsAlphaUTF8 (const UCharT *c);
```

**Arguments**

| | |
|---|---|
| `c` | The character to check |

**Returns**

A nonzero value if the character is a FrameMaker control character, or `0` if it isn't

**Example**

The following code prints `The character is a control character`

```
#include "fencode.h"
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_CharIsEolUTF8(&"\x07"))/* decimal value of 0x07 is 7*/
     F_Printf(NULL,"The character is a control character");
...
```

## F_DigitValue()

Returns the numerical value of a UTF-8 character that is a decimal numeric

**NOTE:** This function doesn't return the value of numeric characters like Ⅳ (code point 0x2163) that aren't in a decimal system.

**Synopsis**

```
#include "fencode.h"
. . .
IntT F_DigitValue (UCharT *s);
```

**Arguments**

| | |
|---|---|
| `s` | The character whose numeric value must be determined |

**Returns**

The numeric value (`0-9`) of the character if it is numeric, or `-1` if it isn't

**Example**

The following code prints ४ + २ = 6

```
...
StringT devanagiri_four="\xE0\xA5\xAA";
StringT devanagiri_two ="\xE0\xA5\xA8";
IntT res;
F_FdeInitFontEncs((ConStringT)"UTF-8");
res = F_DigitValue(devanagiri_four) +F_DigitValue(devanagiri_two);
F_Printf(NULL,"%C + %C is %d", devanagiri_four, devanagiri_two, res);
...
```

### F_IsValidUTF8()

Determines whether the specified byte sequence is a valid UTF-8 string

**Synopsis**

```
#include "fencode.h"
. . .
BoolT F_IsValidUTF8 (ConStringT s);
```

**Arguments**

| | |
|---|---|
| `s` | The string to check |

**Returns**

A nonzero value if the byte sequence is a valid UTF-8 string, or `0` if it isn't

**Example**

The following code prints `First is valid, Second is invalid`

```
#include "fencode.h"
. . .
StringT first = "\x41\xE2\x80\x93\x42";
StringT second = "\x41\xE2\x80";
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_IsValidUTF8(first))
     F_Printf(NULL,"First is valid");
if (!F_IsValidUTF8(first))
     F_Printf(NULL,"Second is invalid");
...
```

### F_UTF8CharSize()

Returns the number of bytes taken by the UTF-8 character based on its first byte.

UTF-8 encoding allows you to determine the number of bytes a character occupies by looking at the first byte of the character. A Unicode character in UTF-8 occupies 1 to 4 bytes .

**Synopsis**

```
#include "fencode.h"
. . .
IntT F_UTF8CharSize (const UCharT c);
```

**Arguments**

| | |
|---|---|
| `c` | The first byte of the character whose size must be found |

**Returns**

The number of bytes occupied by the UTF-8 character (`1-4`)

**Example**

The following code prints `ऄ occupies 3 bytes`

```
#include "fencode.h"
. . .
StringT devanagiri_four="\xE0\xA5\xAA";
IntT n;
F_FdeInitFontEncs((ConStringT)"UTF-8");
n = F_UTF8CharSize(*devanagiri_four);
F_Printf("%C occupies %d bytes", devanagiri_four, n);
...
```

## F_UTF16CharSize()

Returns the number of UTF-16 code units taken by the UTF-16 character based on the first code unit of the character.

UTF-16 encoding allows you to determine the number of code units a character occupies by looking at the first code unit of the character. A UTF-16 code unit is 2 bytes. A Unicode character in UTF-16 occupies 1 to 2 UTF-16 code units.

**Synopsis**

```
#include "fencode.h"
. . .
IntT F_UTF16CharSize (const UChar16T c);
```

**Arguments**

| | |
|---|---|
| `c` | The first code unit of the character whose size must be found |

**Returns**

The number of code units occupied by the UTF-16 character (`1-2`)

**Example**

The following code prints ऄ occupies 1 UTF-16 code units

```
#include "fencode.h"
. . .
StringT devanagiri_four="\xE0\xA5\xAA";
UChar16T devanagiri_four_U16[2];
IntT n;

F_FdeInitFontEncs((ConStringT)"UTF-8");
F_CharUTF8ToUTF16(devanagiri_four, devanagiri_four_U16);
n = F_UTF16CharSize(*devanagiri_four_U16);
F_Printf("%C occupies %d UTF-16 code units", devanagiri_four, n);
...
```

## F_UTF8NextChar()

Increments the UTF-8 character pointer to the next character in the string.

You must use this function to traverse a UTF-8 string. Using `cp++` to parse a string, where `cp` is a character pointer, is incorrect for UTF-8 because it traverses the string on a byte-by-byte basis, rather than a character-by-character basis. This function also returns the incremented pointer.

**Synopsis**

```
#include "fencode.h"
. . .
const UCharT * F_UTF8NextChar (const UCharT **c);
```

**Arguments**

| | |
|---|---|
| `c` | Pointer to the character pointer that must be incremented |

**Returns**

The incremented character pointer

**Example**

The following code prints Бог-Б-о-г

```
#include "fencode.h"
. . .
StringT russian="\xD0\x91\xD0\xBE\xD0\xB3";
UCharT *t=russian;
F_FdeInitFontEncs((ConStringT)"UTF-8");
F_Printf(NULL, russian);
while(*t)
{
     F_Printf(NULL,"-%C",t);
     F_UTF8NextChar(&t);
}
...
```

## F_UTF16NextChar()

Increments the UTF-16 character pointer to the next character in the string.

You must use this function to traverse a UTF-16 string. Using `cp++` to parse a string, where `cp` is a character pointer (`UChar16T` pointer), is incorrect for UTF-16 because it traverses the string on a code-unit-by-code-unit basis, rather than a character-by-character basis. This function also returns the incremented pointer.

**Synopsis**

```
#include "fencode.h"
. . .
const UChar16T * F_UTF16NextChar (const UChar16T **c);
```

**Arguments**

| | |
|---|---|
| `c` | Pointer to the character pointer that must be incremented |

**Returns**

The incremented character pointer.

**Example**

The following code prints `Skipping 2 characters we get r`

```
#include "fencode.h"
. . .
UCharT russian_char_U8[4];
UChar16T russian_U16[]={0x0411,0x043E,0x0433,0x0000};
UChar16T *t=russian_U16;

F_FdeInitFontEncs((ConStringT)"UTF-8");
F_UTF16NextChar(&t);
F_UTF16NextChar(&t);
F_CharUTF16ToUTF8(t,russian_char_U8);

F_Printf(NULL, "Skipping 2 characters we get %C", russian_char_U8);

...
```

## F_StrLenUTF16()

Returns the length of a null-terminated UTF-16 string in terms of number of non-null code units (number of `UChar16T`).

**NOTE:** A UTF-16 string is considered null-terminated if it has a null `UChar16T` (2 bytes), rather than a null `UCharT` (1 byte).

**Synopsis**

```
#include "fencode.h"

. . .

IntT F_StrLenUTF16 (const UChar16T *string);
```

**Arguments**

| | |
|---|---|
| `string` | The string whose length must be determined |

**Returns**

The number of non-null code units (`UChar16T`) in the null-terminated UTF-16 string

**Example**

The following code prints `The length of the string is 3 code units`

```
#include "fencode.h"
. . .
UChar16T russian_U16[]={0x0411, 0x043E, 0x0433, 0x0000};
IntT n;
F_FdeInitFontEncs((ConStringT)"UTF-8");
n = F_StrLenUTF16(russian_U16);
F_Printf(NULL, "The length of the string is %d code units", n);
...
```

## F_CharUTF32ToUTF8()

Converts a UTF-32 character to a UTF-8 character

**Synopsis**

```
#include "fencode.h"
. . .
IntT F_CharUTF32ToUTF8 (UChar32T src, UCharT *dest);
```

**Arguments**

| | |
|---|---|
| `src` | The character to convert |
| `dest` | Pointer to the converted character. It must point to a block of modifiable memory large enough to store the converted character (a UTF-8 character requires at most 4 bytes). |

**Returns**

The number of bytes (or UTF-8 code points) written in `dest`

**Example**

The following code prints `1Б2о3г`

```
#include "fencode.h"
. . .
UChar32T russian_U32[]={0x0411, 0x043E, 0x0433, 0x0000};
UCharT dest[5];
IntT i;

F_FdeInitFontEncs((ConStringT)"UTF-8");
for(i=0;i<3;i++)
     {
     dest[F_CharUTF32ToUTF8(russian_U32[i],dest)]=0;
     F_Printf("%d%s",i,dest); /* note dest is null terminated here */
     }
...
```

## F_CharUTF8ToUTF32()

Converts a UTF-8 character to a UTF-32 character

**Synopsis**

```
#include "fencode.h"
. . .
```

```
UChar32T F_CharUTF8ToUTF32 (const UCharT *src);
```

**Arguments**

| | |
|---|---|
| `src` | The character to convert |

**Returns**

The character in UTF-32

**Example**

The following code prints `0x411`

```
#include "fencode.h"
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
F_Printf("%x",F_CharUTF8ToUTF32("\xD0\x91"));
...
```

## F_CharUTF16ToUTF8()

Converts a UTF-16 character to a UTF-8 character

**Synopsis**

```
#include "fencode.h"
. . .
IntT F_CharUTF16ToUTF8 (const UChar16T *src, UCharT *dest);
```

**Arguments**

| | |
|---|---|
| `src` | The character to convert |
| `dest` | Pointer to the converted character. It must point to a block of modifiable memory large enough to store the converted character (a UTF-8 character requires at most 4 bytes). |

**Returns**

The number of bytes (or UTF-8 code points) written in `dest`

**Example**

The following code prints 1Б2о3г

```
#include "fencode.h"
. . .
UChar16T russian_U16[]={0x0411, 0x043E, 0x0433, 0x0000};
UChar16T *t = russian_U16;
UCharT dest[5];
IntT i;

F_FdeInitFontEncs((ConStringT)"UTF-8");
while(*t)
     {
     F_CharUTF16ToUTF8(t,dest);
     F_Printf("%d%C",i,dest); /* dest is not null terminated here*/
     F_UTF16NextChar(&t);
     }
...
```

## F_CharUTF8ToUTF16()

Converts a UTF-8 character to a UTF-16 character

**Synopsis**

```
#include "fencode.h"

. . .

IntT F_CharUTF8ToUTF16 (const UCharT *src, UChar16T *dest)
```

**Arguments**

| | |
|---|---|
| `src` | The character to convert |
| `dest` | Pointer to the converted character. It must point to a block of modifiable memory large enough to store the converted character (a UTF-16 character requires at most 2 code units of 2 bytes each). |

**Returns**

The number of `UChar16T` (or UTF-16 code points) written in `dest`

**Example**

The following code prints `1,0x411`

```
#include "fencode.h"
. . .
UChar16T russian_U16[2];
IntT n;
F_FdeInitFontEncs((ConStringT)"UTF-8");

n = F_CharUTF8ToUTF16("\xD0\x91",russian_U16);
F_Printf("%d,%x", n, russian_U16[0]);
...
```

IntT F_CharUTF32ToUTF16 (UChar32T src, UChar16T *dest)

UChar32T F_CharUTF16ToUTF32 (const UChar16T *src)

## F_StrAlphaToIntUnicode()

Converts a UTF-8 alphanumeric string into an integer

**NOTE:** This function won't work for numeric characters like Ⅳ (code point 0x2163) that aren't in a decimal system.This function can take decimal digits from mixed decimal systems (6 ૪ , where ૪ is 4 in devanagiri, is valid and interpreted as 64).

**Synopsis**

```
#include "fencode.h"
. . .
IntT F_StrAlphaToIntUnicode (ConStringT string);
```

**Arguments**

| | |
|---|---|
| `string` | The string to convert |

**Returns**

The integer equivalent of the specified string

**Example**

The following code prints ४२ + २४ = 66

```
...
StringT devanagiri_four="\xE0\xA5\xAA";
StringT devanagiri_two ="\xE0\xA5\xA8";
UCharT forty2[256];
UCharT twenty4[256];
IntT res;

FontEncIdT feId = F_FdeInitFontEncs((ConStringT)"UTF-8");
F_StrTruncEnc(forty2,0,feId);
F_StrTruncEnc(twenty4,0,feId);

F_StrCpy(forty2, devanagiri_four);
F_StrCat(forty2, devanagiri_two);
F_StrCpy(twenty4, devanagiri_two);
F_StrCat(twenty4, devanagiri_four);

res = F_StrAlphaToIntUnicode(forty2) + F_StrAlphaToIntUnicode(twenty4);
F_Printf(NULL,"%s + %s is %d", forty2, twenty4, res);
...
```

## F_StrAlphaToRealUnicode()

Converts a UTF-8 alphanumeric string into a `PRealT`

**NOTE:** This function won't work for numeric characters like Ⅳ (code point 0x2163) that aren't in a decimal system. This function can take decimal digits from mixed decimal systems (6. ४ , where ४ is 4 in devanagiri, is valid and interpreted as 6.4).

**Synopsis**

```
#include "fencode.h"

. . .

PRealT F_StrAlphaToRealUnicode (ConStringT string);
```

**Arguments**

| | |
|---|---|
| `string` | The string to convert |

**Returns**

The `PRealT` corresponding to the specified string

**Example**

The following code prints ४ . २ + २ . ४ = 6.6

```
...
StringT devanagiri_four="\xE0\xA5\xAA";
StringT devanagiri_two ="\xE0\xA5\xA8";
UCharT forty2[256];
UCharT twenty4[256];
PRealT res;

FontEncIdT feId = F_FdeInitFontEncs((ConStringT)"UTF-8");
F_StrTruncEnc(forty2,0,feId);
F_StrTruncEnc(twenty4,0,feId);

F_StrCpy(forty2, devanagiri_four);
F_StrCat(forty2, ".");
F_StrCat(forty2, devanagiri_two);
F_StrCpy(twenty4, devanagiri_two);
F_StrCat(twenty4, ".");
F_StrCat(twenty4, devanagiri_four);

res = F_StrAlphaToRealUnicode(forty2)+ F_StrAlphaToRealUnicode(twenty4);
F_Printf(NULL,"%s + %s is %f", forty2, twenty4, res);
...
```

## F_StrCmpUTF8Locale()

Compares the UTF-8 string based on Unicode Collation Algorithm (UCA)

**NOTE:** Because the comparison honors the current system locale, it varies slightly with language rules in different locales.

This doesn't perform a code point-based comparison, but instead uses UCA and localization information in order to compare strings correctly. For example, if used for sorting, it sorts all digits together, sorting the Devanagiri representation of 2 between the English representations of 1 and 3. Similarly, the 'Double Width B' sorts between 'A' and 'C' even though its code point is much higher. This also performs canonical normalization to ensure that the Unicode character sequences `0x00C4` and `0x0041 0x0308`, which are both ways of representing Ä, are considered equivalent.

### Synopsis

```
#include "fencode.h"

. . .

IntT F_StrCmpUTF8Locale (ConStringT s1, ConStringT s2);
```

**Arguments**

| | |
|---|---|
| `s1` | A string |
| `s2` | A string to compare to `s1` |

**Returns**

An integer greater than `0` if `s1` is lexicographically greater than `s2`; `0` if the strings are equal; or an integer less than `0` if `s1` is less than `s2`

**Example**

The following code prints the string `The strings Ä and Ä are equal`

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
StringT s1 = F_StrCopyString("\x41\xCC\x88");
StringT s2 = F_StrCopyString("\xC3\x84");
if(!F_StrCmpUTF8Locale(s1, s2))
     F_Printf(NULL, "The strings %s and %s are equal\n", s1, s2);
. . .
```

## F_StrICmpUTF8Locale()

Compares the UTF-8 string based on Unicode Collation Algorithm (UCA) case insensitively

**NOTE:** Because the comparison honors the current system locale, it varies slightly with language rules in different locales.

This doesn't perform a code point-based comparison, but instead uses UCA and localization information in order to compare strings correctly. For example, if used for sorting, it sorts all digits together, sorting the Devanagiri representation of 2 between the English representations of 1 and 3. Similarly, the 'Double Width B' sorts between 'a' and 'C' even though its code point is much higher. This also performs canonical normalization to ensure that the Unicode character sequences `0x00C4` and `0x0041 0x0308`, which are both ways of representing Ä, are considered equivalent.

**Synopsis**

```
#include "fencode.h"

. . .

IntT F_StrICmpUTF8Locale (ConStringT s1, ConStringT s2);
```

**Arguments**

| | |
|---|---|
| `s1` | A string |
| `s2` | A string to compare to `s1` |

**Returns**

An integer greater than `0` if `s1` is lexicographically greater than `s2`; `0` if the strings are equal; or an integer less than `0` if `s1` is less than `s2`

**Example**

The following code prints the string `a<B<c`

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
StringT B = "\xEF\xBC\xA2";/* double width B */

if ((F_StrICmpUTF8Locale("a", B)<0)
     && (F_StrICmpUTF8Locale(B, "c")<0))
           F_Printf(NULL, "a<B<c");
. . .
```

## F_StrICmpNUTF8Char

Compares the Unicode code points of each character in two UTF-8 strings up to a specified number of UTF-8 characters (not bytes). This function ignores cases.

**Synopsis**

```
#include "fencode.h"

. . .

IntT F_StrICmpNUTF8Char ( ConStringT s1,  ConStringT s2, IntT n);
```

**Arguments**

| | |
|---|---|
| `s1` | A string |
| `s2` | A string to compare to `s1` |

**Returns**

An integer greater than `0` if `s1` is lexicographically greater than `s2`; `0` if the strings are equal; or an integer less than `0` if `s1` is less than `s2`.

**Example**

The following code prints the string `First 2 chars of` Бог `and` Боо `are equal`

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
StringT s1 = F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xB3");
StringT s2 = F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xBE");
if(!F_StrICmpNUTF8Char(s1, s2, 2))
     F_Printf(NULL, "First 2 chars of %s and %s are equal\n", s1, s2);
. . .
```

## F_StrIEqualNUTF8Char ()

Compares the Unicode code points of each character in two UTF-8 strings up to a specified number of UTF-8 characters (not bytes). This function ignores cases.

### Synopsis

```
#include "fencode.h"

. . .

BoolT F_StrIEqualNUTF8Char ( ConStringT s1,  ConStringT s2,  IntT n);
```

### Arguments

| | |
|---|---|
| `s1` | A string |
| `s2` | A string to compare to `s1` |

### Returns

`True` if the strings are equal, `False` otherwise

### Example

The following code prints the string `First 2 chars of` Бог `and` Боо `are equal`

```
   . . .
   F_FdeInitFontEncs((ConStringT)"UTF-8");
   StringT s1 = F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xB3");
   StringT s2 = F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xBE");
   if(!F_StrIEqualNUTF8Char(s1, s2, 2))
        F_Printf(NULL, "First 2 chars of %s and %s are equal\n", s1, s2);

. . .
```

## F_StrCpyNUTF8Char ()

Copies a string to another string, limiting the target string to a specified number of characters

**IMPORTANT:** *The length of the resulting string is calculated in characters, not bytes. This is important because a UTF-8 character may take up to 4 bytes. The resulting string must have adequate space.*

### Synopsis

```
#include "fencode.h"

. . .

IntT F_StrCpyNUTF8Char (StringT s1,ConStringT s2,IntT n);
```

**Arguments**

| | |
|---|---|
| s1 | A string |
| s2 | A string to copy to s1 |
| n | The length limit (in characters) for s1 |

**Returns**

Final length of the copied string in terms of characters

**Example**

The following code prints the string The string Бо was copied from Бог

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
StringT s2 = F_StrCopyString("\xD0\x91\xD0\xBE\xD0\xB3");
UCharT s1[256];
s1[0]=0;
F_StrCpyNUTF8Char(s1,s2,2);
F_Printf(NULL, "The string %s was copied from %s\n", s1, s2);
. . .
```

## F_StrCatUTF8CharNByte ()

Appends a UTF8 character to a string, limiting the length of the resulting string to a specified length

**IMPORTANT:** *The length of the resulting string is calculated in bytes, not characters. This is important because a UTF-8 character may take up to 4 bytes.*

**Synopsis**

```
#include "fencode.h"

. . .

IntT F_StrCatUTF8CharNByte ( StringT s, const UCharT *c,  IntT n);
```

**Arguments**

| | |
|---|---|
| s | The string |
| c | The character to append |
| n | The length limit (in bytes) for s1 |

**Returns**

Final length of the copied string in terms of bytes

**Example**

The following code prints the string Бо + г = Бог

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
UCharT s[256];
StringT a = "\xD0\x91\xD0\xBE";
StringT b = "\xD0\xB3";

s[0]=0;
F_StrCpy(s,a);
F_StrCatUTF8CharNByte( s, b, 256);
F_Printf(NULL, "%s + %s = %s\n", a, b, s);
. . .
```

## F_StrReverseUTF8Char ()

Reverses a specified number of characters in a UTF-8 string

**IMPORTANT:** *This reverses the string code-point-by-code-point (rather than byte-by-byte). However, this still results in absurdities like the diaeresis coming before 'a' when it was after 'a' in the original string. Thus äo will become öa if ä has been stored as 'a' followed by the diaresis.*

**Synopsis**

```
#include "fencode.h"

. . .

VoidT F_StrReverseUTF8Char (StringT s, IntT l);
```

**Arguments**

| | |
|---|---|
| `s` | The string to reverse |
| `l` | The number of characters to reverse. If `l` is greater than the length of the string, `F_StrReverse()` reverses all the characters in the string. |

**Returns**

`VoidT`

**Example**

The following code prints the string ২৩৪ + ৩২৪ = 558

```
StringT devanagiri_four="\xE0\xA5\xAA";
StringT devanagiri_three="\xE0\xA5\xA9";
StringT devanagiri_two ="\xE0\xA5\xA8";
UCharT n_234[256];
UCharT n_324[256];
IntT res;

FontEncIdT feId = F_FdeInitFontEncs((ConStringT)"UTF-8");
F_StrTruncEnc(n_234,0,feId);
F_StrTruncEnc(n_324,0,feId);

F_StrCpy(n_234, devanagiri_two);
F_StrCat(n_234, devanagiri_three);
F_StrCat(n_234, devanagiri_four);

F_StrCpy(n_324,n_234);
F_StrReverseUTF8Char(n_324,2);

res = F_StrAlphaToIntUnicode(n_234) + F_StrAlphaToIntUnicode(n_324);
F_Printf(NULL,"%s + %s = %d", n_234, n_324, res);
...
```

## F_StrStripUTF8Chars ()

Removes a specified set of UTF-8 characters from a UTF-8 string

**IMPORTANT:** *This function doesn't perform canonical or compatibility normalizations. Because it works on a code-point-by-code-point basis, it treats 'a' followed by 'diaeresis' as separate entities despite their forming the grapheme cluster ä.*

If `"äo"` must be stripped from `"göat"` and ä (and ö) are stored as `'a'` (and `'o'`) followed by `'diaeresis'`, the result is `"gt"` because `'a'`, `'diaeresis'` and `'o'` are all removed from the string. If, however, ö is stored in its composed form, the result is `"göt"`. If both ö and ä are stored in their composed form, the result is `"göat"`.

**Synopsis**

```
#include "fencode.h"

. . .

VoidT F_StrStripUTF8Chars (StringT s, ConStringT strip);
```

**Arguments**

| | |
|---|---|
| `s` | The string from which to remove characters |
| `strip` | The characters to remove from `s` |

**Returns**

`VoidT`

**Example**

The following code prints the string `Have you göt a göat?`

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
StringT orig = F_StrCopyString("g\xC3\xB6\xC1\x74");
StringT strip = "a\xCC\x88\x6F"; /* CC 88 is diaeresis */
UCharT cop[256];
cop[0]=0;
F_StrCpy(cop,orig);
F_StrStripUTF8Chars(cop, strip);
F_Printf(NULL, "Have you %s a %s?\n", cop, orig);
. . .
```

## F_StrStripUTF8String ()

Removes all occurrences of one string from another

**IMPORTANT:** *Because this function doesn't perform canonical or compatibility normalizations, it treats 'a' followed by 'diaeresis' as different from the precomposed form ä (code point* `0x00E4`*).*

Removing `"hel"` from `"hehello"` results in `"helo"` and not `"o"`. That is, this performs only one pass.

**Synopsis**

```
#include "fencode.h"

. . .

VoidT F_StrStripUTF8String (StringT s, ConStringT strip);
```

**Arguments**

| | |
|---|---|
| `s` | A string |
| `strip` | The string whose occurrences must be removed from `s` |

**Returns**

`VoidT`

**Example**

The following code prints the string `helo`

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
StringT orig = F_StrCopyString("hehello");
StringT strip = "hel";
F_StrStripUTF8Chars(orig, strip);
F_Printf(NULL, orig);
. . .
```

## F_StrStripUTF8Strings ()

Removes all occurrences of one set of strings from another

**IMPORTANT:** *Because this function doesn't perform canonical or compatibility normalizations, it treats 'a' followed by 'diaeresis' as different from the precomposed form ä (code point* `0x00E4`*).*

**Synopsis**

```
#include "fencode.h"

. . .

VoidT F_StrStripUTF8Strings (StringT s, StringListT sstrip);
```

**Arguments**

| | |
|---|---|
| `s` | A string |
| `sstrip` | The list of strings whose occurrences must be removed from `s` |

**Returns**

`VoidT`

**Example**

The following code prints the string `Vision is a daydream. Action is a nightmare.`

```
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
StringT orig = F_StrCopyString("Vision without action is a daydream.
Action without vision is a nightmare.");
StringListT list;
list = F_StrListNew((UIntT)1, (UIntT)1);
F_StrListAppend(list, (StringT)"action ");
F_StrListAppend(list, (StringT)"without ");
F_StrListAppend(list, (StringT)"vision ");

F_StrStripUTF8Strings(orig, list);
F_Printf(NULL, orig);
. . .
```

### F_StrTokUTF8 ()

Is similar to `F_StrTok` except that it considers UTF-8 characters and not only single bytes. It returns the text tokens in a specified UTF-8 string, separated by occurrences of any combination of one or more of the UTF-8 characters in another string.

**Synopsis**

```
#include "fencode.h"

. . .

StringT F_StrTokUTF8 (StringT s1, ConStringT s2);
```

**Arguments**

| | |
|---|---|
| `s1` | The string to parse for text tokens |
| `s2` | A string containing characters used to separate the text tokens |

**Returns**

The first time you call `F_StrTokUTF8()`, it returns a pointer to the beginning of the first token in `s1` and overwrites `s1` with `NULL` at the end of the token. `F_StrTokUTF8()` keeps track of its position in the string after each call so that when you call it again with `s1` set to `NULL`, it starts immediately after the previous token. The separator string `s2` can be different for each call. When it has passed all the tokens in `s1`, `F_StrTokUTF8()` returns `NULL`.

**Example**

The following code prints the string `4 plus ₹ p 3 + 10 ⊕ 23 = 42`

```
. . .
StringT s, s1,cop, tokens;
IntT i = 0;
F_FdeInitFontEncs("UTF-8");

s1 = F_StrCopyString("4 plus \xE0\xA5\xA8 p 3 + 10 \xE2\x8A\x95 23");
cop = F_StrCopyString(s1);

tokens = " +plus\xE2\x8A\x95";
s = F_StrTokUTF8(s1, tokens);

while (s != NULL)
{
i += F_StrAlphaToIntUnicode(s);
s = F_StrTok(NULL, tokens);
}
F_Printf(NULL, "%s = %d",cop,i);
...
```

## F_StrBrkUTF8 ()

Returns a pointer to the first occurrence in UTF-8 string `s1` of any UTF-8 character in `s2`.

**Synopsis**

```
#include "fencode.h"
. . .
StringT F_StrBrkUTF8 (StringT s1, ConStringT s2);
```

**Arguments**

| | |
|---|---|
| `s1` | The string to search |
| `s2` | A string containing characters to search for |

**Returns**

A pointer to the first occurrence in `s1` of any character in `s2`, or `NULL` if none of the characters in `s2` occurs in `s1`.

**Example**

The following code prints the string `1 ⊕ 2 ⊕ 3 - 2 ⊕ 3 - 3`

```
. . .
StringT s, s1,cop, tokens;
IntT i = 0;
F_FdeInitFontEncs("UTF-8");

s1 = F_StrCopyString("1 \xE2\x8A\x95 2 \xE2\x8A\x95 3");
F_Printf(NULL, s1);
s = F_StrBrkUTF8(s1, "\xE2\x8A\x95")
F_Printf(NULL, " - %s",s);
s = F_StrBrkUTF8(s1, "\xE2\x8A\x95")
F_Printf(NULL, " - %s",s);
...
```

# Modified FDE functions

## F_FdeInitFontEncs

This call now accepts `"UTF-8"` as an input. Calling this with `"UTF-8"` as a parameter enables *Unicode Mode* for the FDE. Calling this function with any other encoding disables *Unicode Mode* and enables *Compatibility Mode* for the FDE. For more information, read the sections on *Unicode Mode* and *Compatibility Mode* earlier in the document.

## F_Printf

This function can now accept the `%C` escape sequence. In *Compatibility Mode*, this ignores the corresponding parameter. In *Unicode Mode*, the first UTF-8 character in the corresponding parameter (which must be `ConStringT` or `UCharT *`) is printed. The following code prints ४ + २ = 6

```
...
StringT devanagiri_four="\xE0\xA5\xAA";
StringT devanagiri_two ="\xE0\xA5\xA8";
IntT res;
F_FdeInitFontEncs((ConStringT)"UTF-8");
res = F_DigitValue(devanagiri_four) +F_DigitValue(devanagiri_two);
F_Printf(NULL,"%C + %C is %d", devanagiri_four, devanagiri_two, res);
...
```

## F_FontEncName, F_FontEncId

These functions can now handle UTF-8. For example, the following code prints `UTF-8`

```
...
FontEncIdT feId;
F_FdeInitFontEncs((ConStringT) "UTF-8");
feId = F_TextEncToFontEnc(F_EncUTF8);
if (feId == F_FontEncId("UTF-8"))
     F_Printf(NULL, F_FontEncName(feId));
...
```

**NOTE:** These functions behave the same in *Unicode Mode* and *Compatibility Mode*.

# New FDE functions

## F_SetICUDataDir()

Sets the ICU data directory for the calling process.

**NOTE:** This function can accept the path in ASCII only. The path must be on a local, a mapped drive or a network path.

**NOTE:** ICU data directory is set on a per-process basis. Certain types of clients, like synchronous DLL clients on Windows, reside in the same process space as FrameMaker. Such clients do not need to set the ICU data directory since FrameMaker sets the ICU data directory for its process. If such clients set the ICU data directory incorrectly using this function, the entire FrameMaker process and other clients might get affected. Such clients should, therefore, be careful while setting the ICU data directory.

**Synopsis**

```
#include "fencode.h"
. . .
VoidT F_SetICUDataDir (ConStringT path);
```

**Arguments**

| | |
|---|---|
| `path` | The ICU Data Directory path (absolute path, not relative) |

**Returns**

`VoidT`

**Example**

The following code sets the ICU data directory of a client to the ICU data directory shipped with FrameMaker 8:

```
...
UCharT icu_path[256];
StringT icu_dir="\icu_data";
StringT fminit_path = F_ApiGetString(0, FV_SessionId, FP_FM_InitDir);
UCharT *t,*s;

s=fminit_path;
while(*s)
     *t++ = *s++;
s=icu_dir;
while(*s)
     *t++ = *s++;
*t=0;

F_SetICUDataDir((ConStringT) icu_path);
...
```

## F_GetICUDataDir()

Gets the currently set ICU data directory for the calling process

**NOTE:** ICU data directory is set on a per-process basis. Certain types of clients, like synchronous DLL clients on Windows, reside in the same process space as FrameMaker. Such clients do not need to set the ICU data directory since FrameMaker sets the ICU data directory for its process. Thus, for such clients, `F_GetICUDataDir` will return the ICU data directory set for the FrameMaker process and therefore will return non-null even if the directory has not explicitly been set using `F_SetICUDataDir` in the client.

**NOTE:** The path returned by this call must not be freed.

**Synopsis**

```
#include "fencode.h"

. . .

ConStringT F_GetICUDataDir (VoidT);
```

**Arguments**

`VoidT`

**Returns**

The currently set ICU data directory path for the calling process

**Example**

The following code sets the ICU data directory of a client to `C:\icu_data` if it isn't set already:

```
...
ConStringT icu_dir = F_GetICUDataDir();
if (!icu_dir || !*icu_dir)
     F_SetICUDataDir((ConStringT) "C:\\icu_data");
...
```

## F_FdeEncodingsInitialized()

Determines whether the FDE encoding data has been correctly initialized.

If this doesn't return true, `F_FdeInit` or `F_FdeInitFontEncs` has not been called or has failed and the FDE won't function correctly. Use this function, along with `F_GetICUDataDir`, to check that the FDE has been correctly set up before making FDE calls.

### Synopsis

```
#include "fencode.h"

. . .

BoolT F_FdeEncodingsInitialized (VoidT);
```

### Arguments

`VoidT`

### Returns

A nonzero value if FDE font encoding has been initialized by `F_FdeInit` or `F_FdeInitFontEncs`, or `0` if it hasn't.

### Example

The following code prints `Font Encoding Data Initialized`

```
#include "fencode.h"
. . .
F_FdeInitFontEncs((ConStringT)"UTF-8");
if (F_FdeEncodingsInitialized())
     F_Printf(NULL,"Font Encoding Data Initialized");
...
```

## F_StrConvertEnc(), F_StrConvertEnc_IgnoreControlChars(), F_StrConvertEnc_ConvertControlChars()

Utility functions for conversion between various encodings

**Synopsis**

```
#include "fencode.h"
. . .
StringT F_StrConvertEnc (ConStringT in,
                    FTextEncodingT from, FTextEncodingT to);
StringT F_StrConvertEnc_IgnoreControlChars (ConStringT in,
                    FTextEncodingT from, FTextEncodingT to);
StringT F_StrConvertEnc_ConvertControlChars (ConStringT in,
                    FTextEncodingT from, FTextEncodingT to);
```

`F_StrConvertEnc` is identical to `F_StrConvertEnc_IgnoreControlChars`. `F_StrConvertEnc_IgnoreControlChars` doesn't modify the lower 32 characters of `0x00-0x1F` and copies them byte-by-byte irrespective of `from` and `to` encoding (except in the case of conversion between `UTF-16` and another encoding, where `0x0010` is converted to `0x10` and vice-versa). `F_StrConvertEnc_ConvertControlChars` does modify the lower 32 characters of `0x00-0x1F` assuming them to be in the `from` encoding. Thus it converts `0x14` to `0x2003` if `from` is `F_EncMakerRoman` and to is `F_EncUTF8`.

UTF-16 strings are expected in the endianness of the platform and are returned in the platform endianness. You must ensure that the input string `in` is valid in the encoding `from`. Incorrect or lossy conversions due to incorrect input strings, incorrect encodings, and conversion limitations inherent to the encodings might cause question marks `'?'` to appear in the string. An empty or NULL string might also be returned. For incorrect UTF-8 inputs (when `from` is `F_EncUTF8`), a string with 3 question marks `"???"` is returned.

**NOTE:** Strings returned by this function must be freed by using `F_Free`

**NOTE:** If `from` is `F_EncUTF16`, the `UChar16T` pointer pointing towards the UTF-16 string should be typecasted to `ConStringT` (it would be treated correctly inside). If `to` is `F_EncUTF16`, the return value should be typecasted to `UChar16T *`.

**NOTE:** `F_EncSpecialSymbol`, `F_EncSpecialZapfDingbats` and `F_EncSpecialWingdings` can only be used as `from` encodings.

**Arguments**

| | |
|---|---|
| `in` | The string to convert |
| `from` | The encoding from which to convert |
| `to` | The encoding to which the string must be converted |

The possible values of the FTextEncodingT parameters from and to are:

| FTextEncodingT | Comments |
|---|---|
| F_EncMakerRoman | Corresponds to FrameRoman encoding |
| F_EncISOLatin1 | |
| F_EncASCII | |
| F_EncANSI | |
| F_EncMacASCII | |
| F_EncJIS7 | |
| F_EncShiftJIS | Corresponds to JISX0208.ShiftJIS encoding |
| F_EncJIS8_EUC | |
| F_EncBig5 | Corresponds to BIG5 encoding |
| F_EncCNS_EUC | |
| F_EncGB8_EUC | Corresponds to GB2312-80.EUC encoding |
| F_EncHZ | |
| F_EncKSC8_EUC | Corresponds to KSC5601-1992 encoding |
| F_EncUTF8 | |
| F_EncUTF16 | |
| F_EncSpecialSymbol | Can only be used as from encoding |
| F_EncSpecialZapfDingbats | Can only be used as from encoding |
| F_EncSpecialWingdings | Can only be used as from encoding |

**Returns**

The converted string

**Example**

The following code prints Symbol αβχ Shift-JIS あぶい

```
#include "fencode.h"
. . .
StringT symb="abc";
StringT sjis="\x82\xA0\x82\xD4\x82\xA2";
StringT temp;

F_FdeInitFontEncs((ConStringT)"UTF-8");
F_Printf(NULL, "Symbol ");
temp = F_StrConverEnc(symb, F_EncSpecialSymbol, F_EncUTF8);
F_Printf(NULL, temp);
F_Free(temp);

F_Printf(NULL, " Shift-JIS");
temp = F_StrConverEnc(sjis, F_EncShiftJIS, F_EncUTF8);
F_Printf(NULL, temp);
F_Free(temp);
...
```

## F_TextEncToFontEnc()

Converts from `FTextEncodingT` to `FontEncIdT`

**NOTE:** This function has different behaviors for *Compatibility Mode* and *Unicode Mode*. In *Unicode Mode*, if no match is found, UTF-8 is the default output. In *Compatibility Mode*, the default output is FrameRoman.

**Synopsis**

```
#include "fencode.h"
. . .
FontEncIdT F_TextEncToFontEnc (FTextEncodingT textEnc);
```

**Arguments**

| | |
|---|---|
| `textEnc` | The text encoding to convert |

**Returns**

The font encoding corresponding to the text encoding

**Example**

The following code prints "JISX0208.ShiftJIS"

```
#include "fencode.h"
. . .
StringT fname = F_FontEncName(F_TextEncToFontEnc(F_EncShiftJIS));
F_FdeInitFontEncs((ConStringT) "UTF-8");
F_Printf(NULL, fname);
...
```

## F_FontEncToTextEnc()

Converts from FontEncIdT to FTextEncodingT

**NOTE:** This function has different behaviors for *Compatibility Mode* and *Unicode Mode*. In *Unicode Mode*, if no match is found, UTF-8 is the default output. In *Compatibility Mode*, the default output is FrameRoman.

**Synopsis**

```
#include "fencode.h"
. . .
FTextEncodingT F_FontEncToTextEnc (FontEncIdT feId);
```

**Arguments**

| | |
|---|---|
| feId | The font encoding to convert |

**Returns**

The text encoding corresponding to the font encoding

**Example**

The following code converts the string myText from the current Dialog Encoding to UTF-8 and prints it on the console:

```
#include "fencode.h"
. . .
F_FdeInit();
F_FdeInitFontEncs("UTF-8");/* Sets FDE to run in Unicode Mode */
F_ApiEnableUnicode(False); /* Sets APIs to run in Compatibility Mode */
myText = F_ApiGetString(...); /* will return in Dialog Encoding dlgEnc*/

encName = F_ApiGetString(0, FV_SessionId, FP_DialogEncodingName);
FTextEncodintT dlgEnc = F_FontEncToTextEnc(F_FontEncId(encName));
convertedText = F_StrConvertEnc(myText, dlgEnc, F_EncUTF8);

F_Printf(NULL, "%s", convertedText); /* convertedText is in UTF-8 */
...
```

# Special characters

## Special handling for lower 32 characters

The lower 32 characters of any encoding `0x00-0x1F` are used by FrameMaker as special control characters. These are different from ASCII at places. For example, `FC_EOL` or `0x09` is the hard-return character in FrameMaker, which is different from the usual `0x0A` used for the end-of-line character. The lower 32 characters are used uniformly across all encodings. Therefore, EM SPACE, which has the standard Unicode representation of `0x2003`, is represented by `0x14` even in UTF-8 in the context of FDK APIs. These character mappings can be seen in `fcharmap.h` in the FDK include folder.

As a result, some strings returned from APIs aren't strictly in UTF-8 format. In order to convert entirely to UTF-8, any character below 32 would have to be appropriately mapped to a UTF-8 character (for example `0x14` would be mapped to the Unicode code point `0x2003`). Similarly, the APIs do not accept strings truly in UTF-8 format. The character `0x2003` won't behave as EM SPACE in FrameMaker unless it is first converted to `0x14` before being passed to an API. The deviation from Unicode is only for the lower 32 characters.

## Sensitivity of certain calls towards special characters

Certain calls, especially FilePath and I/O functions in the FDE and `F_ApiOpen`, `F_ApiSave`, and `F_ApiImport`, that deal with filepaths are more sensitive towards the presence of certain special characters (below 32 characters) than they were in earlier releases.

For example, APIs like `F_ApiOpen` that were relatively insensitive to the presence of `'\r'` (`CR`) and `'\n'` (`LF`) characters no longer work if these characters are present in the filename. It is the client's responsibility to ensure that such special characters are stripped before calling APIs and FDE functions.

## Unicode equivalents of special characters

The header file `fcharmap.h` defines characters like `FC_DAGGER` that are in FrameRoman encoding. Equivalent Unicode characters (in UTF-16 format) have also been added in the header file. These have been suffixed with a `_U` to indicate that they are the Unicode equivalents of previously defined characters.

| **Character** | ***Code Point*** | ***Description*** |
|---|---|---|
| `FC_UTILITY_U` | `0x01` | used by search and index |
| `FC_DBREAK_U` | `0x02` | discretionary break |

| | | |
|---|---|---|
| FC_NBREAK_U | 0x03 | suppress this break |
| FC_DHYPHEN_U | 0x04 | discretionary hyphen |
| FC_NHYPHEN_U | 0x05 | suppress this h-point |
| FC_HYPHEN_U | 0x06 | temporary hyphen |
| FC_TAB_U | 0x08 | |
| FC_EOL_U | 0x09 | hard return |
| FC_EOP_U | 0x0A | end of para |
| FC_EOD_U | 0x0B | end of flow |
| FC_SPACE_NUMBER_U | 0x10 | number space |
| FC_SPACE_HARD_U | 0x11 | hard space |
| FC_SPACE_THIN_U | 0x12 | thin == 1/12 em |
| FC_SPACE_EN_U | 0x13 | en == 1/2 em |
| FC_SPACE_EM_U | 0x14 | em == 1 em |
| FC_HYPHEN_HARD_U | 0x15 | unbreakable explicit hyphen |
| FC_ESC_U | 0x1B | sentinel code for cblocks |
| FC_SCH_U | 0x1C | sentinel code for sblocks |
| FC_SPACE_U | 0x20 | ' ' regular space |
| FC_QUOTEDBL_U | 0x22 | "" straight double quote |
| FC_QUOTESINGLE_U | 0x27 | '\'' straight single quote |
| FC_BACKSLASH_U | 0x5c | '\\' backslash |
| FC_GUILLEMOTLEFT_U | 0x00AB | guillemotleft |
| FC_GUILLEMOTRIGHT_U | 0x00BB | guillemotright |
| FC_QUOTELEFT_U | 0x2018 | curly single-left quote |
| FC_QUOTERIGHT_U | 0x2019 | curly single-right quote |
| FC_QUOTEDBLLEFT_U | 0x201C | curly double-left quote |
| FC_QUOTEDBLRIGHT_U | 0x201D | curly double-right quote |

| | | |
|---|---|---|
| FC_GUILSINGLLEFT_U | 0x2039 | guillemotleft |
| FC_GUILSINGLRIGHT_U | 0x203A | guillemotright |
| FC_QUOTESINGLBASE_U | 0x201A | quotesinglbase |
| FC_QUOTEDBLBASE_U | 0x201E | quotedblbase |
| FC_CENT_U | 0x00A2 | |
| FC_POUND_U | 0x00A3 | |
| FC_YEN_U | 0x00A5 | |
| FC_ENDASH_U | 0x2013 | |
| FC_DAGGER_U | 0x2020 | |
| FC_DAGGERDBL_U | 0x2021 | |
| FC_BULLET_U | 0x2022 | |
| FC_EMDASH_U | 0x2014 | |
| FC_META_U | 0x80 | |