Multilingual Text Editor API Preliminary Documentation

For Multilingual Text Editor 1.1

Preliminary Draft

Technical Publications

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for typographical errors.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Important

> This is a preliminary document. Although it has been reviewed for technical accuracy, it is not final. Apple Computer, Inc. is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation.
>
> You can check <http://developer.apple.com/techpubs/macos8/SiteInfo/whatsnew.html> for information about updates to this and other developer documents. To receive notification of documentation updates, you can sign up for ADC's free Online Program and receive their weekly Apple Developer Connection News e-mail newsletter. (See <http://developer.apple.com/membership/index.html> for more details about the Online Program.)

Executive Summary

This document describes the engineering requirements for a new multilingual text editor (MLTE). As a text-editing engine, MLTE is intended for use by applications that aren’t primarily oriented towards word processing or page layout. MLTE provides sufficient built-in functionality for applications with simple-to-midlevel text-editing needs.

The MLTE is intended as an alternative for TextEdit, the basic text-editing engine in the Mac OS. All reasonable developer-requested enhancements to TextEdit (such as document-wide tabs, full justification, and support for more than 32K of text) are supported by MLTE. MLTE does not offer API compatibility with TextEdit. MLTE does offer equivalent or greater functionality than TextEdit. MLTE provides the API to build a complete text editing user experience as defined in *Macintosh Human Interface Guidelines*, the Drag and Drop Human Interface Guidelines, and *Inside Macintosh: Text.*

MLTE uses Apple Type Services for Unicode Imaging (ATSUI) to measure and draw text if ATSUI is available. If ATSUI is not available, then MTLE uses QuickDraw and the Script Manager to handle text. MLTE can run on systems back to System 7.1.

With MLTE, layout settings (i.e., tabs, justification, are margins) are document wide.

MTLE supports 32 levels of undo. In addition, the can undo and can redo functions return a key to the type of user action that can be undone or redone. It is the callers responsibility to map these keys to the appropriate localized string to display to the user. Actions that can be undone are listed below in the section in the Data Structures and Constants section.

MTLE also supports the saving and opening of files that are:
- plain text
- plain text with commonly supported style resources
- plain Unicode text

- a new format that supports either text or Unicode text along with embedded graphics, sounds, and movies.

Architecture

Aspects of Long-term Architecture

The primary goal is to provide a text-editing engine that provides a level of basic functionality higher than that offered by TextEdit and supports editing Unicode™ text. This is the case for basic editing tasks and for the level of multilingual text editing. MLTE will also provide an API that is expandable, and much more easily modified than the TextEdit API. To this end, opaque data structures are used to encapsulate all data used by MLTE.

Fit with Apple System Architecture

MLTE requires CFM as a dynamic linking mechanism. MLTE will be a step in providing world-ready text editing with sufficient functionality to cover most developer needs. This will further developers in creating single code bases for delivering products to multiple international markets.

On systems prior to system 8.6, MLTE is a client of QuickDraw Text and the Script Manager. Beginning with system 8.6, ATSUI replaces QuickDraw Text and the Script Manager as the low-level means of imaging and measuring text utilized by MLTE.

Where required, MLTE fully supports the Text Service Manager for text input.

MLTE provides the last significant building block towards creating a Mac OS that uses Unicode for all text.

Features and Behaviors

MTLE supports all languages that currently are supported on the Macintosh and supports inline input for Chinese, Japanese and Korean. MLTE also supports Unicode text, and input methods written for non-CJK scripts if running on system 8.6 or later.

MLTE provides all of the enhancements that developers have requested for TextEdit. These include support for greater than 32K of text and a document wide tab setting. Version 1.0 of MLTE will offer only a single tab setting, but later versions may offer multiple tab settings via rulers.

User Experience

This section specifies the default user experience provided by MLTE. It pays particular attention to the specifics of editing multiscript text, which may involve contextual or bidirectional text layout or using inline input. It expands on specifications given in *the Macintosh Human Interface Guidelines*, the Drag and Drop Human Interface Guidelines, and *Inside Macintosh: Text*.

Text Formatting

MLTE renders text into a single rectangular frame. Applications can choose between assuming arbitrarily wide lines and breaking lines at a certain width. When breaking lines, MLTE uses the simplistic line breaking model that's usually used on the Macintosh: that is, text is flowed into a visual line as long as it fits, then a new line is started with the first unbreakable unit (e.g., word) that didn't completely fit into the line. In scripts that use space characters to separate words, one (and only one) space character at the logical end of the text flowed into a visual line is consumed by the line break – it is ignored for measurements and not displayed. This last description only applies when using QuickDraw. It ATSUI is used, line break and display is controlled by the ATSUI line breaking algorithms.

The interpretation of Tab characters is based on the one-tab-per-document standard found in most programming text editors. Each tab character maps to an initial width. As text is flowed onto a line, each tab is replaced by the width value necessary to place the start of the text following the tab at a given position on the line. As the text prior to the tab grows, the white space appears to shrink until the preceding text becomes long enough to envelop the entire tab. At that point, the tab will assume its full width and the text following the tab will jump ahead. The following illustration will help to clarify this point.

Figure 1:

<<Intial state white space between text block A and text block B represents a tab>>

text block a   text block b

<<user enters text in text block a>>

text block a with more   text block b

<<text in block a reaches a length that displaces the beginning of block b>>

text block a with more text   text block b

The tab widths flow in the line direction for the line being formatted. If text is being automatically wrapped and a tab width extends past the trailing margin (right on a Roman system), a line break is generated and the next visual line will begin with the tab width.

Justification in version 1.0 of the MLTE might more appropriately be called *flush*. Text can be flush against the left margin, flush against the right margin, centered or flush against both margins (typically referred to as *full justification*).

Highlight regions for non-empty selections are drawn in the system highlight color, while carets are drawn in black.

For bidirectional text, the caret location at direction run boundaries depends on the direction of the keyboard script; split carets are not supported. Outline highlighting is used for inactive views as required for Drag and Drop. For non-modifiable text that allows for selections, an application can choose between one of two behaviors. The first allows selection and copying of text and displays a blinking caret. This is the MPW model. The second type of non-modifiable behavior is to display no caret and not allow selection. This is the Simple Text model.

Selection Behavior

Selection behavior is described in *Macintosh Human Interface Guidelines*, pages 286-296, with details on Arrow keys in *Macintosh Human Interface Guidelines*, pages 281-284. The specification given here has been adjusted to more closely correspond to the de facto standard for text selection found in the more popular text editors used on the Macintosh.

A user has two ways to define a selection: she can create a new one or modify the current one. A new selection is defined by the Select All command, by mouse actions (single-, double-, or triple-clicking or dragging), or by using the Arrow keys (potentially combined with the Command or Option keys). A selection is modified by pressing the Shift key and performing a mouse-based or Arrow key-based selection action.

MLTE interprets modifying selection actions based on the notions of anchor selection and active selection, implementing what's called the fixed-point method in *Macintosh Human Interface Guidelines*, page 290. The active selection is (with one exception – see below) identical to the selection resulting from the non-modifying selection action that would be performed without the Shift key. The anchor selection is the result of a previous selection action, it is updated whenever the user creates a new selection, edits the text, deactivates the view, or when the selection is changed through an API call. The modified selection is the smallest selection containing both the anchor selection and the active selection.

When tracking mouse down events, MLTE automatically disambiguates between selection operations and Drag and Drop operations. If the mouse down event occurs within the highlight region of the current selection and the Drag Manager is available, then MLTE waits to see whether the mouse is dragged. If it is, MLTE initiates a Drag and Drop operation. Drag and Drop behavior is discussed below. Otherwise, the mouse event is interpreted as a selection operation.

Single-clicking defines an insertion point. Double-clicking by default selects a word as defined by the Script Manager or ATSUI. Triple-clicking selects a visual line from the beginning of the line to the beginning of the next line. If the user starts selecting by dragging after a double or triple click, dragging extends the selection by words or visual lines, respectively. Clicking in empty space is mapped to some location that has text.

The Arrow key in page direction (down for Roman) starts at the screen location of the logical end of the current selection and simulates successive clicks in each line moving in page direction in as straight a line as possible. The Arrow key against page direction (up for Roman) starts at the screen location of the logical start of the current selection and simulates successive clicks in each line moving straight against the page direction.

Horizontal Arrow keys move in a direction dependent on the line direction of the text. The Arrow key in line direction (right for Roman) starts at the trailing edge of the highlight region in the last line of the selection and simulates successive clicks at each character boundary moving in line direction until it hits the trailing edge of the visual line. At that point selection wraps to the leading edge of the next visual line. The character boundaries are determined by the backing store order and not the display order.

The Arrow key against line direction (left for Roman) starts at the leading edge of the highlight region in the first line of the selection and simulates successive clicks at each character boundary moving against line direction until it hits the leading edge of the visual line, then wraps to the trailing edge of the previous visual line.

For the line direction Arrow keys, a ligature that does not allow for an insertion point between its constituting characters is treated as one character. This may be controllable in an environment with ATSUI. Combining the Option key with a line direction Arrow key makes it simulate clicks at word boundaries instead of character boundaries. The implementation of Option-Arrow is slightly different from that recommended on *Macintosh Human Interface Guidelines*, page 296. The guidelines state that pressing option and either Left Arrow or Right Arrow should select the entire word. The MLTE implementation is to select from the insertion point (anchor point) to either the beginning or end of the word where the insertion point resides.

Combining the Command key with an Arrow key in or against line direction makes it simulate clicks at the trailing-or-leading edge of the last-or-first line intersecting with the selection, respectively. When reaching a direction run boundary, a click on the last character in Arrow direction of the direction run being left is simulated; the direction run being entered is clicked on only after the direction run boundary has been passed.

Combining the Command key with a page direction Arrow key makes it simulate a click at the corresponding edge of the portion of the view shown in the window, paging the view first if the active selection already was at that edge. The start selection for page direction Arrow keys is determined at the beginning of an uninterrupted sequence of page direction Arrow keys.

The active selection is initially determined by an action defining a new selection and then updated by each modifying selection action. If a modifying selection action results in an active selection that is a subrange of the anchor selection, the active selection is set to the subrange. The exception to the rule above – that the active selection of a modifying selection action is equal to the selection that would have been created by the same action without the Shift key – are Arrow keys in or against line direction. In that case, if the current selection is not empty and the Shift key is not held down, they first simulate a click on the trailing-or-leading edge of the highlight region. If the Shift key is held down, they immediately simulate a click one character apart from that edge.

If necessary, the text is scrolled to make a modified selection visible in the view rectangle.

Selection actions never result in the system beeping at the user.

Typing and Inline Input

MLTE treats text entry using standard keyboard layouts and text entry using input methods in an integrated fashion. An uninterrupted sequence of keystrokes or inline input operations are treated as a single typing command for purposes of Undo. Events that cause a typing command to be completed include: selection operations (except for those handled by input methods), deactivation, filing, printing, or any undoable command other than typing. When a typing command is completed, any unconfirmed inline input is confirmed.

MLTE assumes that the application filters out all characters it wishes to handle before passing key-down events to MLTE. MLTE then interprets the characters being entered in the following ways: (Note that the rules below are given for both Unicode (Uxxx) and Mac OS encodings ($xx)).

• Insertion: All 1- and 2-byte characters starting at ($20, U0020) except Forward Delete ($7F, U007F), as well as the Tab character ($09, U0009), and the characters $10-$14 (which are graphical characters in some fonts, especially system fonts) are inserted into the text. Return ($0D, U000D) is inserted. Characters entered through inline input are always inserted.

• Select: All combinations involving the Arrow keys ($1C-$1F, U001C-U001F) are interpreted as selection operations (see above). Other than as specified in *Macintosh Human Interface Guidelines*, page 113, they do interrupt typing commands in MLTE.

• Scroll: The Home ($01, U0001) and End ($04, U0004) keys are interpreted to scroll the text block to its logical beginning or end as specified in *Macintosh Human Interface Guidelines*, page 285.

• The Page Up ($0B, U000B) and Page Down ($0C, U000C) keys: These are interpreted to scroll the text one up or down by the height of the currently visible portion. They are not part of typing commands, but don’t interrupt them either.

• Clear: The Clear key (character code $1B,U001B with virtual key code $47) is a synonym for the Clear command. It is not part of a typing sequence, but does interrupt one.

• Delete: The Backspace ($08,U0008) and Forward Delete ($7F, U007F) characters first delete the currently selected text (if the selection is non-empty), then delete individual characters logically preceding (Backspace) or following (Forward Delete) the insertion point. They are part of typing commands.

• Ignore: All other characters are ignored. This includes all key combinations involving the Command key, but not Arrow keys. They are not part of typing commands, but do not interrupt them.

Keyboard and Font Synchronization

In a multiscript environment, a text engine has to make sure that text is displayed in a font that supports the character set in which the text is written. In the WorldScript environment, this is typically done by watching the current keyboard script and comparing it to the script of the font at the current insertion point. If the two don’t match and the user starts typing, the font is automatically replaced with one belonging to the keyboard script.

This behavior is not always appropriate, as there is no one-to-one correspondence between fonts and keyboards. Typically, non-Roman keyboard layouts support only the characters that are specific to this script, not the ASCII characters which are supported by all fonts designed for the WorldScript environment. Thus, when the user switches to a Roman keyboard, she may do so just to type ASCII characters, and the previously used non-Roman font may have glyphs for the ASCII characters that are carefully designed to match the style of the other glyphs in the font, making it highly undesirable to replace them with plain Geneva. In addition, a Unicode font may contain glyphs that apply to multiple scripts.

Despite these drawbacks, MLTE will by default attempt to synchronize the font to the keyboard when the user changes the keyboard. To find the appropriate font, MLTE first searches backward in the document for an appropriate font, then forward. If no appropriate font can be found, the application font or the system font for the keyboard script is used. Font synchronization does not interrupt typing commands.

Font to Keyboard Synchronization

Some editors also support synchronization in the opposite direction: they automatically switch the keyboard script to the script of the font being used at the current selection under certain circumstances; for example, when the user changes the selection. The assumption is that the user is most likely to type additional text in the script already being used for the current selection. Also, the location of the caret in bidirectional text may depend on the direction of the keyboard script, so in this context it is important that the direction of the keyboard script matches the direction of text in which the user clicked.

Many users of 2-byte systems strongly dislike this feature. In 2-byte scripts, the issue of caret placement doesn’t exist, and 2-byte input methods often allow users to enter ASCII characters in a pass-through mode, so switching the keyboard is not necessary. Users of 1-byte scripts on the other hand can enter ASCII characters only by switching to a Roman keyboard.

The current plan is to support font to keyboard synchronization by default. There will be an option for an application to switch font to keyboard synchronization off.

Font Locking

By default, MLTE prevents a user from changing a font in one script to a font in another. Version 1.0 will maintain this behavior. However, a user can override this by changing the font while holding the control key down. In this case, the text will change to the selected font no matter what characters is selected. In addition, when a user selects non-Roman text and changes the text to a roman font, the text is scanned for ASCII characters, and these characters are changed to the new font.

Drag and Drop

If the Drag Manager is available, MLTE provides a large part of the Drag and Drop user experience as specified in the Drag and Drop Human Interface Guidelines. MLTE highlights selections in inactive views using outlines, so users can drag between active and inactive views. It changes the cursor to an arrow if it is over the highlight region in an active view. It disambiguates between selection operations and Drag and Drop operations, and provides the complete drag user feedback. Because MLTE has no information about the context in which it's views are used, it cannot provide complete destination feedback, but it does highlight the insertion point where dropped text would get inserted, performs the actual move, and selects the dragged text in its new location.

By Default MLTE recognizes dragging as a Move operation. The user can override the move-or-copy decision using the Option Key.

Drag-and-drop operations (both move and copy) are undoable.

Support for Standard Editing Menus

While MLTE does not handle any menus itself, it provides applications with all necessary functionality and information to support the standard text editing menus.

MLTE supports the Undo, Redo, Cut, Copy, Paste, Clear, and Select All items in the Edit menu, as specified in *Macintosh Human Interface Guidelines*, pages 109-117. MTLE does not support Publish and Subscribe.

MLTE supports the specifications in *Macintosh Human Interface Guidelines*, pages 120-122 and pages 64-67, for the Font menu. Because of the large difference in font environments on a system with ATSUI and a QuickDraw system, there is an API that builds a Font menu and returns that menu to the application. The application will be able to call another API to correctly handle font menu selection via the returned font menu.

MLTE supports the specifications in *Macintosh Human Interface Guidelines*, pages 122-123 and pages 64-67, for the Size menu, including the use of checkmarks and dashes, increment size, decrement size, and an Other item.

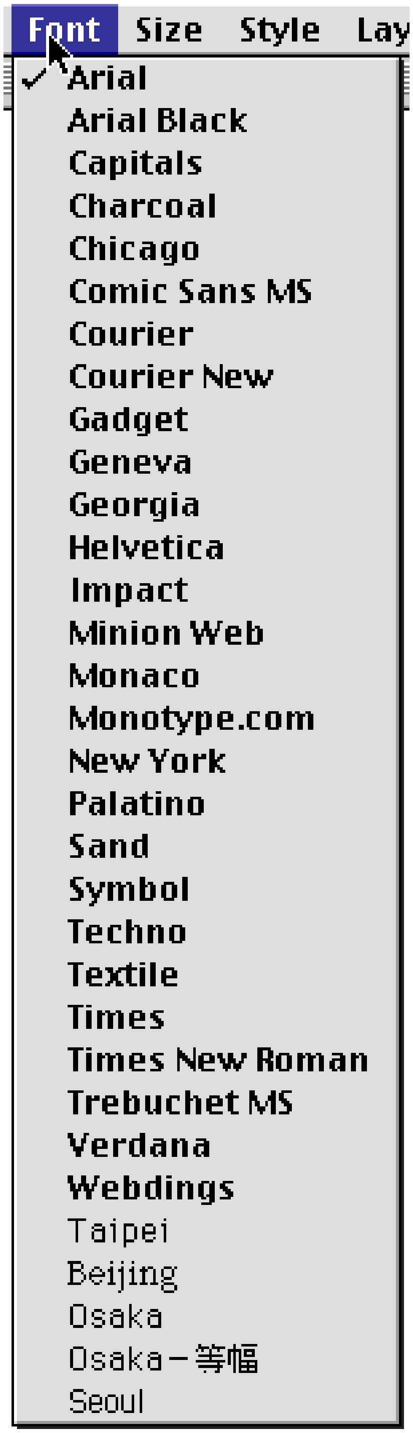
MLTE supports the specifications in *Macintosh Human Interface Guidelines*, pages 124 and 64-67, for the Style menu, including the use of checkmarks and dashes.

Cut, Copy, Paste, Clear, are undoable commands. Applying a font, size, or style to an non-empty selection is an undoable command, while applying them to an insertion point is not. Select All is a selection operation and is not undoable. All commands mentioned here interrupt typing commands.

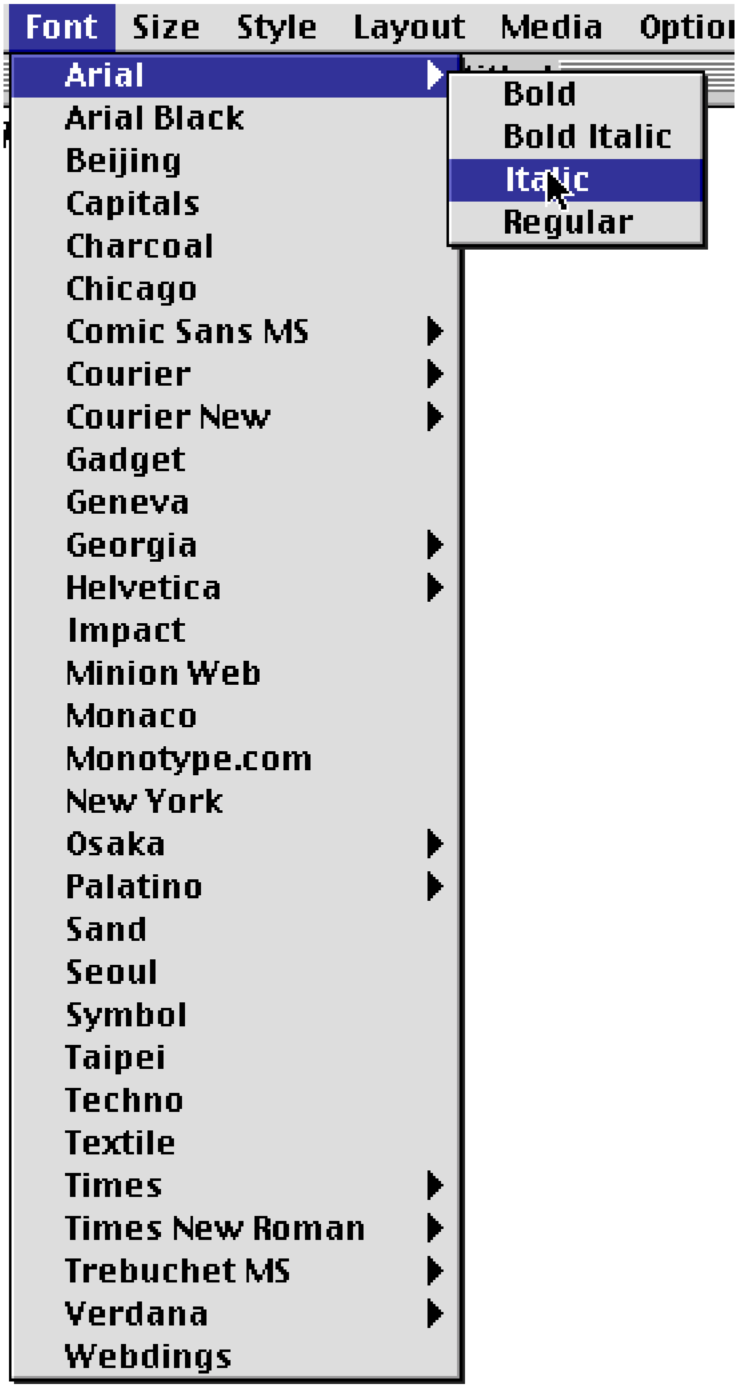
Font Menu

Because MLTE supports both QuickDraw and ATSUI without requiring applications to know which is being used, it becomes difficult to leave the responsibility for building the font menu to the application. Certainly, there are applications who would prefer to build their own, and there is nothing to prevent them from doing so. For applications that would prefer not to have to bother with the issues of building a font menu, MLTE provides utility functions for creating and handling a standard font menu (where standard is defined as what is most appropriate for the imaging system in use.)

If the application is running MLTE on a QuickDraw-only system, the Font menu will represent each font with a single item. Fonts will be sorted by script, and will be drawn in the appropriate system font based on the script system that the font belongs to. The following illustration is of an MLTE Font menu on a QuickDraw system. The menu item names will be the font family resource names. In other words, an MLTE Font menu on a QuickDraw-only system will look exactly like a font created today with the AddResMenu call.



MLTE on a system equipped with ATSUI will build a font menu that includes hierarchical sub-menus for ATSUI fonts that share a family name, but have different style names. Each font menu item will be drawn in a single System font, because the concept of script systems is not entirely appropriate in a Unicode world. The following illustration is of an MLTE font menu on a system running ATSUI.



Font menu item names will be names obtained by calling the ATSUI function ATSUGetFontName.

MLTE provides an opaque structure called TXNFontMenuObject that can be used to handle user interaction with the font menu.

ATSUI Font Variations and Features

ATSUI also introduces the concept of font variations and font features. An application can pass these through the MLTE API, and have them applied to a selection. This, like building the font menu, requires that an application be aware of the fact that it's running on an ATSUI system, and further requires that the application use some of the moderately complicated ATSUI API. If an application wishes to do this, it is entirely appropriate, but applications who want to provide basic editing may not be interested in interacting directly with the system's lower-level text imaging software. At the same time such applications may want to offer users at least some of the advanced capabilities in software like ATSUI.

Version 1.0 of the MLTE does not provide a human interface for allowing a user to view and select font variations and features on a per font basis. Unfortunately, this capability is left to the application or enhancements to the system software.

Intelligent Editing

Version 1.0 will not support Intelligent Editing. Intelligent Editing means applying text-modifying commands so that separate words are kept separate and duplicate space characters are avoided.

Key Algorithms

MLTE text handling is based on the layout algorithms found in the Script Manager and the text imaging provided by QuickDraw Text. When ATSUI is available, text handling is based on on the layout algorithms in ATSUI. Text and style runs are accessed and stored as arrays.

Compatibility

MLTE is fully compatible with all Script systems, encodings, and languages currently supported by the Script Manager. It is compatible with all systems since 7.1 and all PPC CPUs. It is also compatible with the Unicode encoding as supported by ATSUI.

MLTE is not compatible with 68K systems. (Additionally, ATSUI is not compatible with 68K systems.)

Internationalization

MLTE is dependent on the Script Manager, ATSUI and WorldScript I and II for laying out text. It is international to the extent that these components are international.

Fault Handling Methodology and Mechanisms

The primary failure encountered by MLTE is lack of memory for adding or formatting data. When this occurs, the operation is not performed and an error is returned to the application.

To a large degree, preflighting is used to prevent error conditions that cannot be backed out again. Since errors are eventually bubbled up to the application, it is the application's responsibility to alert the user to the problem. If the user continues to try and add data, MLTE will just continue to not perform the requested addition and return the same error.

Application Programming Interface (API) for MLTE

***Data Structures and Constants**.*

```
typedef struct OpaqueTXNObject*            TXNObject;
```

An opaque structure that encapsulates an object containing private variables and functions necessary to handle text formatting at a document level. For each document, a new TXNObject is allocated and returned by the TXNNewObject function.

```
typedef struct OpaqueTXNFontMenuObject*  TXNFontMenuObject;
```

An opaque structure that contains information needed work with a font menu.

```
typedef UInt32  TXNFrameID;
```

A TXNFrameID is used to identify the text frame to which actions should be applied. At the basic level there is only one frameID per document. In version 1.0 of MLTE, TXNFrameID serves as a placeholder to permit multiple frame capability to be added in a future version.

```
typedef UInt32                                  TXNVersionValue;
typedef OptionBits                              TXNFeatureBits;
enum {
      kTXNWillDefaultToATSUIBit      = 0
};

enum {
      kTXNWillDefaultToATSUIMask   =  1L<<kTXNWillDefaultToATSUIBit
};
```

These type definitions and constants are used by the function TXNVersionInformation (see below).

```
typedef OptionBits      TXNInitOptions;

enum {
      kTXNWantMoviesBit             = 0,
      kTXNWantSoundBit              = 1,
      kTXNWantGraphicsBit           = 2,
      kTXNAlwaysUseQuickDrawBit      = 3,
      kTXNUseTemporaryMemoryBit      = 4
};



enum {
      kTXNWantMoviesMask            = 1L << kTXNWantMoviesBit,
      kTXNWantSoundMask              = 1L << kTXNWantSoundBit,
      kTXNWantGraphicsMask           = 1L << kTXNWantGraphicsBit,
      kTXNAlwaysUseQuickDrawMask     = 1L << kTXNAlwaysUseQuickDrawBit,
      kTXNUseTemporaryMemoryMask     = 1L << kTXNUseTemporaryMemoryBit
};
```

TXNInitOptions are passed to the function TXNInitTextension. They specify data types other than text that the application wishes to support for future TXNObjects that are allocated within this context. Additionally, an application can request that MLTE always use QuickDraw even if ATSUI is available. For applications whose biggest concern is speed and efficient memory usage, this is often the best choice. Finally, an application can request that all memory allocations required inside the MLTE text engine should use memory from temporary memory.

```
typedef OptionBits            TXNFrameOptions;

enum {
      kTXNDrawGrowIconBit            = 0,
      kTXNShowWindowBit              = 1,
      kTXNWantHScrollBarBit          = 2,
      kTXNWantVScrollBarBit          = 3,
      kTXNNoTSMEverBit               = 4,
      kTXNReadOnlyBit                = 5,
      kTXNNoKeyboardSyncBit          = 6,
      kTXNNoSelectionBit             = 7,
      kTXNSaveStylesAsSTYLResourceBit = 8,
      kOutputTextInUnicodeEncodingBit = 9,
      kTXNDoNotInstallDragProcsBit = 10,
      kTXNAlwaysWrapAtViewEdgeBit    = 11
};



enum {
   kTXNDrawGrowIconMask              = 1L << kTXNDrawGrowIconBit,
   kTXNShowWindowMask                = 1L << kTXNShowWindowBit,
   kTXNWantHScrollBarMask            = 1L << kTXNWantHScrollBarBit,
   kTXNWantVScrollBarMask            = 1L << kTXNWantVScrollBarBit,
   kTXNNoTSMEverMask                 = 1L << kTXNNoTSMEverBit,
   kTXNReadOnlyMask                  = 1L << kTXNReadOnlyBit,
   kTXNNoKeyboardSyncMask            = 1L << kTXNNoKeyboardSyncBit,
   kTXNNoSelectionMask               = 1L << kTXNNoSelectionBit,
   kTXNSaveStylesAsSTYLResourceMask = 1L <<kTXNSaveStylesAsSTYLResourceBit,
   kOutputTextInUnicodeEncodingMask = 1L << kOutputTextInUnicodeEncodingBit,
   kTXNDoNotInstallDragProcsMask = 1L << kTXNDoNotInstallDragProcsBit,
   kTXNAlwaysWrapAtViewEdgeMask = 1L << kTXNAlwaysWrapAtViewEdgeBit
};
```

TXNFrameOptions are used to specify per TXNObject features (i.e. per document features). The available options are:

`kTXNDrawGrowIconMask:` Draw a grown icon at the bottom right corner of the frame.
`kTXNShowWindowMask:` Display the window before returning from TXNNewObject.
`kTXNWantHScrollBarMask:` Include and manage a horizontal scroll bar inside the frame.

`kTXNWantHScrollBarMask:` Include and manage a vertical scroll bar inside the frame.
`kTXNNoTSMEverMask:` This TXNObject should never be TSM aware. This option is not allowed when the text being used is Unicode text since TSM is required for inputting any Unicode character.
`kTXNReadOnlyMask:` Date inside this TXNObject is read-only.
`kTXNNoKeyboardSyncMask:` Do not synchronize the keyboard with the font (see above in User Interface section for further discussion of keyboard synchronization).
`kTXNNoSelectionMask:` Do not display the insertion point.
`kTXNSaveStylesAsSTYLResourceMask:` When saving data has text save style information as 'styl' resources (SimpleText) compatibility.
`kOutputTextInUnicodeEncodingMask:` When saving plain text save it as Unicode.
`kTXNAlwaysWrapAtViewEdgeMask:` Always word-wrap at the edge of the TXNObject's view rectangle.

```
typedef OptionBits                                   TXNContinuousFlags;

enum {
   kTXNFontContinuousBit            = 0,
   kTXNSizeContinuousBit            = 1,
   kTXNStyleContinuousBit           = 2,
   kTXNColorContinuousBit           = 3
};


enum {
   kTXNFontContinuousMask           = 1L << kTXNFontContinuousBit,
   kTXNSizeContinuousMask           = 1L << kTXNSizeContinuousBit,
   kTXNStyleContinuousMask          = 1L << kTXNStyleContinuousBit,
   kTXNColorContinuousMask          = 1L << kTXNColorContinuousBit
};
```

TXNContinuousFlags are passed to the function TXNGetContinuousTypeAttributes. They indicate the type of continuous style information the application is interested in. For the more uncommon style attributes offered by ATSUI, there is another function, TXNGetContinuousTypeTags, which can be used to obtain continuous run information.

```
typedef OptionBits                                   TXNMatchOptions;

enum {
   kTXNIgnoreCaseBit                = 0,
   kTXNEntireWordBit                = 1,
   kTXNUseEncodingWordRulesBit      = 31
};


enum {
   kTXNIgnoreCaseMask               = 1L << kTXNIgnoreCaseBit,
   kTXNEntireWordMask               = 1L << kTXNEntireWordBit,
   kTXNUseEncodingWordRulesMask = 1L << kTXNUseEncodingWordRulesBit
};
```

TXNMatchOptions are passed to the function TXNFind, and specify the matching rules that should be used in the find operation.

```
typedef OSType                                              TXNFileType;

enum {
   kTXNTextensionFile              = FOUR_CHAR_CODE('txtn'),
   kTXNTextFile                    = FOUR_CHAR_CODE('TEXT'),
   kTXNPictureFile                 = FOUR_CHAR_CODE('PICT'),
   kTXNMovieFile                   = MovieFileType,
   kTXNSoundFile                   = FOUR_CHAR_CODE('sfil'),
   kTXNAIFFFile                    = FOUR_CHAR_CODE('AIFF')
};
```

The TXNFileType defines the possible file types that can be passed to the function TXNNewObject.

```
typedef OSType                TXNDataType;

enum {
   kTXNTextData                    = FOUR_CHAR_CODE('TEXT'),
   kTXNPictureData                 = FOUR_CHAR_CODE('PICT'),
   kTXNMovieData                   = FOUR_CHAR_CODE('moov'),
   kTXNSoundData                   = FOUR_CHAR_CODE('snd '),
   kTXNUnicodeTextData             = FOUR_CHAR_CODE('utxt')
};
```

TXNDataType is used in multiple MLTE functions. It is used to specify the type of data being requested or returned.

```
typedef FourCharCode          TXNControlTag;
enum {
      kTXNLineDirectionTag         =          'lndr',
      kTXNJustificationTag         =          'just',
      kTXNIOPrivilegesTag          =          'iopv',
      kTXNSelectionStateTag        =          'slst',
      kTXNInlineStateTag           =          'inst',
      kTXNWordWrapStateTag         =          'wwrs',
      kTXNKeyboardSyncStateTag     =          'kbsy',
      kTXNAutoIndentStateTag       =          'auin',
      kTXNTabSettingsTag           =          'tabs',
      kTXNRefConTag                =          'rfcn',
      kTXNMarginsTag               =          ‘marg’,     //set the top &
                                                          //left margins
      kTXNNoUserIOTag              =          ‘nuio’      //do not allow
                                                          //typing, but do
                                                          //allow
TXNSetData
                                                          //to work
};
```

The type TXNControlTag and its following enumerated constants is used to specify the type of information you are setting or getting when the functions TXNSetTXNObjectControls or TXNGetTXNObjectControls are called.

MLTE returns optional action key codes (i.e. if the caller is not interested a NULL can be passed) in TXNCanUndo and TXNCanRedo. These numeric codes identify the action that can be undone or redone. No strings are involved so MLTE is not concerned with localizing anything. The client is responsible for mapping the key code to an appropriate localized string for user display.

The currently defined action keys are:

```
typedef UInt32    kTXNActionKey;
enum
{
      kTXNTypingAction                          = 0,
      kTXNCutAction                             = 1,
      kTXNPasteAction                           = 2
      kTXNClearAction                           = 3,
      kTXNChangeFontAction                      = 4,
      kTXNChangeFontColorAction                 = 5,
      kTXNChangeFontSizeAction                  = 6,
      kTXNChangeStyleAction                     = 7,
      kTXNAlignLeftAction                       = 8,
      kTXNAlignCenterAction                     = 9,
      kTXNAlignRightAction                      = 10,
      kTXNDropAction                            = 11,
      kTXNMoveAction                            = 12,
      kTXNFontFeatureAction                     = 13,
      kTXNFontVariationAction                   = 14,
      kTXNUndoLastActioon                       = 1024
}

enum {
   kTXNClearThisControl        = (long)0xFFFFFFFF,
   kTXNClearTheseFontFeatures  = (long)0x80000000
};
```

This constants can be used to clear ATSUI control or font feature settings.

The following constant values are used to set the value of a TXNControlData structure before passing that structure to the TXNSetTXNObjectControls or TXNGetTXNObjectControls function.

```
enum {
   kTXNLeftToRight                         = 0,
   kTXNRightToLeft                         = 1
};


enum {
   kTXNFlushDefault                  = 0,/* according to the line direction */
   kTXNFlushLeft                     = 1,
   kTXNFlushRight                    = 2,
   kTXNCenter                        = 4,
   kTXNFullJust                      = 8,
   kTXNForceFullJust                 = 16
};

enum {
   kTXNReadWrite                     = false,
   kTXNReadOnly                      = true
};

enum {
   kTXNSelectionOn                   = true,
   kTXNSelectionOff                  = false
};

enum {
   kTXNUseInline                     = false,
   kTXNUseBottomline                 = true
};

enum {
   kTXNAutoWrap                      = false,
   kTXNNoAutoWrap                    = true
};

enum {
   kTXNSyncKeyboard                  = false,
   kTXNNoSyncKeyboard                = true
};

enum {
   kTXNAutoIndentOff                 = false,
   kTXNAutoIndentOn                  = true
};

typedef Boolean                                   TXNScrollBarState;

enum {
   kScrollBarsAlwaysActive           = true,
   kScrollBarsSyncWithFocus          = false
};
```

The TXNTabType, its enumerated values, and the TXNTab structure are used when calling the TXNSetTXNObjectControls or TXNGetTXNObjectControls function to get tab information for a given TXNObject. Note that in version 1.0 of MLTE only right tabs are supported the other constants are place holders for future enhancements.

```
typedef SInt8                                           TXNTabType;

enum {
   kTXNRightTab                      = -1,
   kTXNLeftTab                       = 0,
   kTXNCenterTab                     = 1
};
```

```
struct TXNTab {
   SInt16                                          value;
   TXNTabType                                      tabType;
   UInt8                                           filler;
};
typedef struct TXNTab                              TXNTab;
```

The TXNTab structure specifies tab information. In the future, three types of tabs may be supported (right, left and center). MLTE 1.0 supports only one left tab per.

```
struct TXNMargins {
   SInt16                                          topMargin;
   SInt16                                          leftMargin;
   SInt16                                          bottomMargin;
   SInt16                                          rightMargin;
};
typedef struct TXNMargins                          TXNMargins;
```

This structure is used to specify the margin value. In version 1.0 of MLTE only the topMargin and leftMargin can be set. BottomMargin and rightMargin are placeholders for future enhancements.

```
union TXNControlData {
      UInt32                                             uValue;
      SInt32                                             sValue;
      TXNTab                                             tabValue;
      TXNMargins *                                       marginsPtr;
};
typedef union TXNControlData TXNControlData;
```

The TXNControlData structure is used to provide or get values from the TXNGetTXNObjectControls and TXNSetTXNObjectControls functions. These functions provide information about any globally set attribute of a TXNObject.

The following constants are convenience definitions used to specify defaults when calling the function TXNSetFontDefaults or to specify that the current type size should decrement or increment by one point when calling the function TXNSetTypeAttributes.

```
enum {
   kTXNDontCareTypeSize        = (long)0xFFFFFFFF,
   kTXNDontCareTypeStyle       = 0xFF,
   kTXNIncrementTypeSize       = 0x00000001,
   kTXNDecrementTypeSize       = (long)0x80000000
};


typedef UInt32                                           TXNOffset;
enum {
   kTXNUseCurrentSelection           =   0xFFFFFFFFUL,
   kTXNStartOffset                   =     0UL,
   kTXNEndOffset                    =     0x7FFFFFFFUL
};
```

TXNOffset is used to specify offsets in a TXNObject's data. kTXNStartOffset and kTXNEndOffset are convenience constants that can be used to specify the start and end of the data in a TXNObject. KTXNUseCurrentSelection can be used to specify that MLTE should just use the current selection.

```
typedef void *                                          TXNObjectRefcon;
```

TXNObjectRefcon is a reference set by MLTE and passed to the filter.

```
enum {
   kTXNShowStart                    = false,
   kTXNShowEnd                      = true
};
```

These constants are passed to TXNShowSelection. They specify whether the application wants the end of the current selection to scroll to be shown or the beginning.

```
typedef FourCharCode                              TXNTypeRunAttributes;

enum {
   kTXNQDFontNameAttribute     = FOUR_CHAR_CODE('fntn'),
   kTXNQDFontFamilyIDAttribute       = FOUR_CHAR_CODE('font'),
   kTXNQDFontSizeAttribute     = FOUR_CHAR_CODE('size'),
   kTXNQDFontStyleAttribute    = FOUR_CHAR_CODE('face'),
   kTXNQDFontColorAttribute    = FOUR_CHAR_CODE('klor'),
   kTXNTextEncodingAttribute   = FOUR_CHAR_CODE('encd')
};

typedef ByteCount
   TXNTypeRunAttributeSizes;

enum {
   kTXNQDFontNameAttributeSize             = sizeof(Str255),
   kTXNQDFontFamilyIDAttributeSize   = sizeof(SInt16),
   kTXNQDFontSizeAttributeSize             = sizeof(SInt16),
   kTXNQDFontStyleAttributeSize            = sizeof(Style),
   kTXNQDFontColorAttributeSize            = sizeof(RGBColor),
   kTXNTextEncodingAttributeSize           = sizeof(TextEncoding)
};
```

The above types and constants are used to set type attributes when calling the function TXNSetTypeAttributes, TXNGetContinuousTypeTags or TXNGetContinuousTypeAttributes. These are supplemented by the style attributes defined for ATSUI.

```
typedef UInt32                 TXNPermanentTextEncodingType;

enum {
   kTXNSystemDefaultEncoding   = 0,
   kTXNMacOSEncoding                 = 1,
   kTXNUnicodeEncoding               = 2
};
```

TXNPermanentTextEncodingType and the accompanying constants are used to specify how the application wants to see text. Specifying one of the specific encodings (kTXNSystemDefaultEncoding, kTXNUnicodeEncoding) means that MLTE will treat all offsets, incoming, and outgoing text as that encoding. This is true even if MLTE is internally dealing with text in another format. If that is the situation MLTE will utilize the Text Encoding Convertor (TEC) to convert text and offsets to match the applications preference. If kTXNSystemDefaultEncoding is specified MLTE will return offsets and text data in the format used internally.

```
typedef FourCharCode                             TXTNTag;

union TXNAttributeData {
   void *                                        dataPtr;
   UInt32                                        dataValue;
};
typedef union TXNAttributeData                   TXNAttributeData;

struct TXNTypeAttributes {
   TXTNTag                                 tag;
   ByteCount                               size;
   TXNAttributeData                        data;
};
typedef struct TXNTypeAttributes           TXNTypeAttributes;
```

The data structures TXTNTag and TXNTypeAttributes are used to request or receive information about the text in a TXNObject.

```
struct TXNATSUIFeatures {
   ItemCount                                     featureCount;
   ATSUFontFeatureType *                   featureTypes;
   ATSUFontFeatureSelector *         featureSelectors;
};
typedef struct TXNATSUIFeatures                  TXNATSUIFeatures;

struct TXNATSUIVariations {
   ItemCount                                     variationCount;
   ATSUFontVariationAxis *                 variationAxis;
   ATSUFontVariationValue *          variationValues;
};
typedef struct TXNATSUIVariations          TXNATSUIVariations;

union TXNAttributeData {
   void *                                        dataPtr;
   UInt32                                        dataValue;
   TXNATSUIFeatures *                      atsuFeatures;
   TXNATSUIVariations *              atsuVariations;
};
typedef union TXNAttributeData                   TXNAttributeData;

struct TXNTypeAttributes {
   TXTNTag                                tag;
   ByteCount                                    size;
   TXNAttributeData                       data;
};
typedef struct TXNTypeAttributes          TXNTypeAttributes;
```

The structures TXNATSUIFeatures, TXNATSUIVariations are used to specify ATSUI font feature or variation setttings when calling the function TXNSetTypeAttributes.

```
struct TXNMacOSPreferredFontDescription {
   UInt32                                       fontID;
   Fixed                                        pointSize;
   TextEncoding                           encoding;
   Style                                        fontStyle;
};
typedef struct TXNMacOSPreferredFontDescription
TXNMacOSPreferredFontDescription;
```

TXNMacOSPreferredFontDescription is used to specify the preferred font for a given text encoding. An array of these structures is passed to TXNInitTextension to specify font defaults for each script.

```
typedef UInt32                                            TXNBackgroundType;

enum {
   kTXNBackgroundTypeRGB            = 1
};


union TXNBackgroundData {
   RGBColor                                     color;
};
typedef union TXNBackgroundData                 TXNBackgroundData;

struct TXNBackground {
   TXNBackgroundType                      bgType;
   TXNBackgroundData                      bg;
};
typedef struct TXNBackground              TXNBackground;
```

A TXNBackground structure is passed to TXNSetBackground to specify the background for text and data in a given TXNObject. At this time only colors are supported.

```
typedef OSStatus                                TXNErrors;
enum {
   kTXNEndIterationErr              = -22000,
   kTXNCannotAddFrameErr            = -22001,
   kTXNInvalidFrameIDErr            = -22002,
   kTXNIllegalToCrossDataBoundariesErr = -22003,
   kTXNUserCanceledOperationErr = -22004,
   kTXNBadDefaultFileTypeWarning = -22005,
   kTXNCannotSetAutoIndentErr  = -22006,
   kTXNRunIndexOutofBoundsErr  = -22007,
   kTXNNoMatchErr                    = -22008,
   kTXNAttributeTagInvalidForRunErr = -22009,
   /*dataValue is set to this per invalid tag*/
   kTXNSomeOrAllTagsInvalidForRunErr = -22010,
   kTXNInvalidRunIndex              = -22011,
   kTXNAlreadyInitializedErr   = -22012,
   kTXNCannotTurnTSMOffWhenUsingUnicodeErr = -22013,
   kTXNCopyNotAllowedInEchoModeErr = -22014
};
```

These errors can be returned by MLTE functions along with memory or file operations.

Functions

```
EXTERN_API( OSStatus )
TXNNewObject
                        (const FSSpec *          iFileSpec, /* can be NULL */
                        WindowPtr                iWindow,
                        Rect *                   iFrame, /* can be NULL */
                        TXNFrameOptions          iFrameOptions,
                        TXNFrameType             iFrameType,
                        TXNFileType              iFileType,
                        TXNPermanentTextEncodingType  iPermanentEncoding,
                        TXNObject *              oTXNObject,
                        TXNFrameID *             oTXNFrameID,
                        TXNObjectRefcon          iRefCon);
```

Allocates a new TXNObject (i.e. the C++ operator new is called to allocate a TXNObject) and returns a pointer to the object in the newDoc parameter.

Input:

iFileSpec: If not NULL, the file is read to obtain the document contents after the object is successfully allocated. If NULL you start with an empty document. Data embedding in not supported by TXNNewObject. If the caller wants to include data that is embedded inside private data they should create the TXNObject by calling TXNNewObject with a NULL iFileSpec. After the TXNObject is created the data can be read in using TXNSetDataFromFile.

iWindow: The window in which the document is going to be displayed. This parameter can also be NULL. If it is NULL, you must eventually attach a Window or Grafport to the TXNObject.

iFrame: If the text-area does not fill the entire window, this specifies the area to fill. If you pass NULL, the window's portRect is used as the frame.

iFrameOptions: Specify the options to be supported by this frame. See the enumerated type TXNFrameOptions for the supported options.

iFileType: Specify the primary file type. If you use kTextensionTextFile, files will be saved in a private format (see xxx). If you want saved files to be plain text files, you should specify 'TEXT' here. If you specify 'TEXT' here, you can use the frameOptions parameter to specify whether the TEXT files should be saved with 'MPSR' resources or 'styl' resources. These are resources that contain style information for a file, and they both have there own limitations. If you use 'styl' resources to save style info, your documents can have as many styles as you like however tabs will not be saved. If you use 'MPSR' resources, only the first style in the document will be saved. (Your application is expected to apply all style changes to the entire document.) If you want media-rich documents that can contain graphics and sound, you should specify kTextensionTextFileOutput. If you want a plain text editor with capabilities similar to SimpleText, specify that style information by saved as ‘styl’ resources. If you want files similar to those output by CW IDE, BBEdit, and MPW, specify that style information be saved in a ‘MPSR’ resource.

iPermanentEncoding: The general encoding(s) that the application considers text to be in. There are three options:
kTXNSystemDefaultEncoding—use the encoding that is preferred by MLTE and the system. This will be Unicode on a system that includes ATSUI.
KTXNMacOSEncoding—incoming and outgoing text should be in traditional MacOS Script system encodings.
kTXNUnicodeEncoding, incoming and outgoing text should be in Unicode even on systems that do not include ATSUI.

Output:

OSStatus: function result. If anything goes wrong, the error is returned. Success must be complete. That is, if everything works, but there is a failure reading a specified file, the object is freed.

oTXNObject: Pointer to the opaque data structure allocated by the function. Most of the subsequent functions require that such a pointer be passed in.

oTXNFrameID: Unique ID for the frame. Although some functions require a TXNFrameID it is for now a placeholder.

```
EXTERN_API( void )
TXNDeleteObject (TXNObject          iTXNObject);
```

Delete a previously allocated TXNObject and all associated data structures.

Input:

`iTXNObject`: opaque structure to free.

```
EXTERN_API( void )
```

```
TXNResizeFrame( TXNObject            iTXNObject,
                UInt32               iWidth,
                UInt32               iHeight,
                TXNFrameID           iTXNFrameID);
```

Changes the frame's size to match the new width and height.
Input:

`iTXNObject`: opaque MLTE structure.

`iWidth`: New width in pixels.

`iHeight`: New height in pixels.

iTXNFrameID: FrameID that specifies the frame to move.

```
EXTERN_API( void )
TXNSetFrameBounds(TXNObject          iTXNObject,
                   SInt32            iTop,
                    SInt32           iLeft,
                    SInt32           iBottom,
                    SInt32           iRight,
                   TXNFrameID        iTXNFrameID);
```

Changes the frame's viewrect to have the new width and height.
Input:
iTXNObject : opaque MLTE structure.

iTop, iLeft, iBottom, iRight: Rect of the view

iTXNFrameID: FrameID that specifies the frame to move.

```
EXTERN_API( OSStatus )
TXNInitTextension(const TXNMacOSPreferredFontDescription  iDefaultFonts[],
                ItemCount            iCountDefaultFonts,
                TXNInitOptions       iUsageFlags);
```

Initialize MLT. Should be called as soon as possible after the Macintosh toolbox is initialized. This function should only be called once per context. If it is called more than once, this function returns a result code of -22012. If this is returned, you can still call other MLTE functions, but any `TXNInitOptions` and `TXNMacOSPreferredFontDescription` specified will not be applied.

Input:

TXTMacOSPreferredFontDescription: A table of font information including fontFamily ID, point size, style, and script code. The table can be NULL or can have an entry for any script for which you would like to to designate a default font. Only a valid script number is required. You can designate that MLTE should use the default for a give script by setting the field to -1.

For example, if you wanted to specify New York as the default font to use for Roman scripts, but were happy with the default style and size, you would call the function like this:

```
TXNMacOSPreferredFontDescription     defaults;
GetFNum( "\pNew York", &defaults.fontFamilyID );
defaults.pointSize = -1;
defaults.fontStyle = -1;
defaults.script = smRoman;
status = TXNInitTextension( &defaults, 1, 0 );
```

usageFlags: Specify whether sound and movies should be supported.

Output:

OSStatus: Function result. NoErr is returned if everything initialized correctly. Variety of possible MacOS errors if something goes wrong.

```
EXTERN_API( void )
TXNTerminateTextension(void);
```

Close the MLTE library. It is necessary to call this function so that MLTE can correctly close down any TSM connections and and do other clean up.

```
EXTERN_API( void )
TXNKeyDown(    TXNObject             iTXNObject,
                const EventRecord *  iEvent);
```

Process a keydown event. Note that if the CJK script is installed and the current font is CJK inline, input will take place. This is always the case unless the application has requested the bottomline window or has turned off TSM (see initialization options above).

Input:

iTXNObject: opaque struct to apply keydown to.

iEvent: the keydown event.

```
EXTERN_API( void )
TXNAdjustCursor( TXNObject           iTXNObject,
                 RgnHandle           ioCursorRgn);
```

Handle switching the cursor. If the mouse is over a text area, set the cursor to the i-beam. If the cursor is over graphics, a sound, a movie, a scroll bar, or outside of window, set the cursor to the arrow cursor.

Input:
iTXNObject: Opaque struct obtained from TXNNewObject.
ioCursorRgn: Region to be passed to WaitNextEvent. Resized accordingly by TXNAdjustCursor.

```
EXTERN_API( void )
TXNClick(   TXNObject                iTXNObject,
            const EventRecord *      iEvent);
```

Processes a mouse-down event in the window's content region. This function takes care of scrolling, selecting text, playing sound and movies, handling drag–and-drop operations, and responding to double-clicks.

Input:
`iTXNObject`: Opaque struct obtained from TXNNewObject.
`iEvent`: the mouse-down event

```
EXTERN_API( Boolean )
TXNTSMCheck( TXNObject               iTXNObject, /* can be NULL */
             EventRecord *           iEvent);
```

Call this when WaitNextEvent returns false or there is no active TSNObject . The TXNObject parameter can be NULL, allowing an application to call this function at any time. This is necessary to ensure input methods enough time to be reasonably responsive.

Input:
`iTXNObject`: The currently active TXNObject or NULL.
`iEvent:` The event record.

Output:
Boolean: True if TSM handled this event. False if TSM did not handle this event.

```
EXTERN_API( void )
TXNSelectAll(                        TXNObject         iTXNObject);
```

Selects all data belonging to the TXNObject.

Input:
iTXNObject: opaque TXNObject

```
EXTERN_API( void )
TXNFocus(   TXNObject                 iTXNObject,
            Boolean                   iBecomingFocused);
```

Focuses the TXNObject. By default, scroll bars and the insertion caret are made active if iBecomingFocused is true, and inactive if false. However, in conjunction with TXNActivate scroll bars can remain active even though text input is not focussed. This is handy for windows containing multiple text areas that are scrollable.

Input:

`iTXNObject`:   opaque TXNObject
`iBecomingFocused`:   true if becoming active. false otherwise.

```
EXTERN_API( void )
TXNUpdate(  TXNObject                 iTXNObject);
```

Handles an update event (i.e. draw everything in a frame.) This function calls the Toolbox BeginUpdate - EndUpdate functions for the window that was passed to TXNNewObject. This makes it inappropriate for windows that contain something else besides the TXNObject. In that case, applications should use TXNDraw to update TXNObjects (see below.)

Input:
`iTXNObject`: opaque TXNObject

```
EXTERN_API( void )
TXNDraw(   TXNObject                  iTXNObject,
           GWorldPtr                  iDrawPort);
```

Redraw the TXNObject including any scroll bars associated with the text frame. Call this function in response to an update event for a window that contains multiple TXNObjects or some other graphic elements. If it is necessary, the application is responsible for calling BeginUpdate/EndUpdate in response to the update event.

Input:
iTXNObject: opaque TXNObject to draw
iDrawPort: This parameter can be NULL. If it is NULL drawing takes place in the port currently attached to the iTXNObject. If not NULL drawing goes to the iDrawPort. This capability can be used to image a TXNObject to a printer as is (i.e without re-layout to a page the printer page size.)

```
EXTERN_API( void )
TXNForceUpdate(      TXNObject         iTXNObject);
```

Force a frame to be updated. This function is of course very much like the toolbox calls InvalRect or InvalRgn.

Input:
iTXNObject: opaque TXNObject

```
EXTERN_API( UInt32 )
TXNGetSleepTicks(TXNObject           iTXNObject);
```

Depending on state of window, get the appropriate sleep time to be passed to WaitNextEvent.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject

Output:
UInt32: function result. The appropriate sleep time.

```
EXTERN_API( void )
TXNIdle(TXNObject                    iTXNObject);
```

Do any necessary Idle time processing. Typically flash the cursor. If a TSMDocument is active, pass a NULL event to the Text Service Manager.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject

```
EXTERN_API( void )
TXNGrowWindow(TXNObject              iTXNObject,
                const EventRecord *    iEvent);
```

If the application has requested a grow region, and if the TXNObject is contained in a window and not a subframe of that window track, then the cursor and grow the TXNObjects view rectangle.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
event: The mouse-down event

```
EXTERN_API( void )
TXNZoomWindow(TXNObject             iTXNObject,
              short                 iPart);
```

Handle mouse-down events in the zoom box. This function should only be called for TXNObject's whose view rect occupies the entire window (e.g., a window is passed to TXNNewObject with a NULL FrameRect.)

Input:
`iTXNObject`: opaque TXNObject obtained
from TXNNewObject
`iPart`: Value returned by FindWindow

```
EXTERN_API( Boolean )
TXNCanUndo(TXNObject                iTXNObject,
           TXNActionKey*            oActionKey);
```

Use this to determine if the Undo item in the Edit menu should be highlighted or not. The result is true if the last command was undoable, and false if it was not undoable.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
Output:
Boolean Function result. If true, the last command is undoable and the undo item in the menu should be active. If false, the last command cannot be undone and undo should be grayed in the menu.
oActionKey: The numeric key which identifies the action that can be undone. The caller of TXNCanUndo is responsible for mapping the key to the appropriate localized string to be displayed to the user.

```
EXTERN_API( Boolean )
TXNCanRedo(TXNObject                iTXNObject,
           TXNActionKey*            oActionKey);
```

Use this to determine if the Redo item in the Edit menu should be highlighted or not. The result is true if the last command was redoable, and false if it was not redoable. The iActionKey identifies the action to be redone. The caller of TXNCanRedo can map the action key to a localized string if the caller wishes to display to the user exactly what can be redone. For example, if the value of iActionKey was kTXNTyping the client could then map that value to a string that read "Redo Typing" on a system localized for U.S. English. Note that MLTE does not supply any mechanisms for doing such a mapping. MLTE simply returns a key that can be used to map to a user readable string that describes the action. All issues of text localization are left to the client of MLTE.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject

Output:

Boolean Function result. If true, the last command is redoable and the redo item in the menu should be active. If false, the last command cannot be redone and redo should be grayed in the menu.

oActionKey: The numeric key which identifies the action that can be redone. The caller of TXNCanRedo is responsible for mapping the key to the appropriate localized string to be displayed to the user(See above for a more complete discussion of how the key might be used).

```
EXTERN_API( void )
TXNUndo (TXNObject                   iTXNObject);
```

Undo the last command. The undo level in MLTE 1.0 is 32 levels deep. That is Undoable actions are collected until the total count is 32. If a user undoes two actions she will need to do redo twice to get back to the original state. If more than 32 actions are performed the oldest actions are forgotten as each new action takes place.

Finally, performing a new action when the last action done was a redo removes any actions currently in a redo state from the stack. For example, say a user performs the following actions: type some text, cut some text, paste some text, type some text; undo the last typing action, and undo the paste operation; redo the paste; type some new text. After the new text has been typed the undo stack will contain: the first text that was typed, the cut action, and the new text that was just typed. The paste action and the second block of typed text will no longer be available for undo, and the new text will be the only action that is undable.

Input:

iTXNObject: An opaque TXNObject obtained from TXNNewObject

```
EXTERN_API( void )
TXNRedo (TXNObject                   iTXNObject);
```

Redo the last command. The undo level in MLTE 1.0 is 1 level deep. That is, if the user undoes an action and then undoes it again, the second undo will be the same as a redo.

Input:

iTXNObject: An opaque TXNObject obtained from TXNNewObject

```
EXTERN_API( OSStatus )
TXNCut (TXNObject                    iTXNObject);
```

Cut the current selection to the MLTE private clipboard. See below for description of clipboard formats.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject

Output:

OSStatus: function result. Variety of memory or scrap MacOS errors.

```
EXTERN_API( OSStatus )
TXNCopy (TXNObject                   iTXNObject);
```

Copy the current selection to the MLTE private clipboard.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
Output:
OSStatus: function result. Memory or parameter errors.

```
EXTERN_API( OSStatus )
TXNPaste (TXNObject                  iTXNObject);
```

Paste the clipboard into the TXNObject.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
Output:
OSStatus: function result. Memory or parameter errors.

```
EXTERN_API( OSStatus )
TXNClear (TXNObject  iTXNObject);
```

Clear the current selection from the TXNObject. Equivalent to selecting something and typing the delete key.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
Output:
OSStatus: function result. Memory or parameter errors.

```
EXTERN_API( void )
TXNGetSelection (TXNObject           iTXNObject,
                 TXNOffset *         oStartOffset,
                 TXNOffset *         oEndOffset);
```

Get the absolute offsets of the current selection. Embedded graphics, sound, etc. each count as one character. Offsets in MLTE are always character offsets.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject
Output:
oStartOffset: absolute beginning of the current selection.
oEndOffset: end of current selection.

```
EXTERN_API( void )
TXNShowSelection (TXNObject          iTXNObject,
                   Boolean           iShowEnd);
```

Scroll the current selection into view.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
iShowEnd: If true, the end of the selection is scrolled into view. If false, the beginning of selection is scrolled into view.

```
EXTERN_API( Boolean )
TXNIsSelectionEmpty (TXNObject       iTXNObject);
```

Call this function to find out if the current selection is empty. Use this to determine if Cut, Copy, and Clear should be highlighted in Edit menu.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
Output:
Boolean: function result. True if current selection is empty (i.e. start offset == end offset). False if selection is not empty.

```
EXTERN_API( OSStatus )
TXNSetSelection (TXNObject           iTXNObject,
                  TXNOffset          iStartOffset,
                  TXNOffset          iEndOffset);
```

Set the current selection. Offset values are character offsets.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
iStartOffset: The new start offset.
iEndOffset: The new end offset.

```
EXTERN_API( OSStatus )
TXNGetContinuousTypeAttributes (TXNObject iTxnObject,
                                                    TXNContinuousFlags *   oContinuousFlags,
                                                    ItemCount              ioCount,
                                                    TXNTypeAttributes      ioTypeAttributes[]);
```

Test the current selection to see if the font, style, color, and/or size of the font is continuous. The flag bits will be set to indicate which of these attributes are continuous. Addtionally, an application can pass in an array for TXNTypeAttributes with the tags set to the continuous attribute that she would like returned. On ATSUI system there is a much larger number of type attributes that might be continuous. TXNGetContinuousTypeAttributes is designed to make it easier for an application to add check marks to the Font, Style, and Size menus. If an application is interested in the other less traditional type attributes available in ATSUI, the call TXNGetContinuousTypeTags should be used instead of TXNGetContinuousTypeAttributes. However, whether MLTE is using QuickDraw or ATSUI to draw text, this function supports size, font, color, and style in either case.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject

continuousFlags: Bits which can be examined to see which if any of the font attributes are continuous. If a particular bit is set and if the application has passed a TXNTypeAttribute in the array that corresponds to the bit, then the information in the TXNTypeAttribute can be used to to do something like check off the continuous size in the size menu.

For example:

```
TXNTypeAttributes      sizeAttr;

sizeAttr.tag = kTXNQDFontSizeAttribute;
sizeAttr.size = kTXNQDFontSizeAttributeSize;
sizeAttr.data.dataValue = 0;

TXNAreFontAttributesContinuous(txnObject, &flags, 1, &sizeAttr);

if ( flags & kSizeContinuousMask )
       CheckSizeMenu( sizeAttr.data.dataValue );
```

ioCount: Count of TXNTypeAttributes records in the ioTypeAttributes array.

ioTypeAttributes: Array of TXNTypeAttributes. The tag values in this array indicate the type attributes the application is interested in.

```
EXTERN_API( OSStatus )
TXNSetTypeAttributes(TXNObject       iTXNObject,
                     ItemCount       iAttrCount,
                     TXNTypeAttributes iAttributes[],
                     TXNOffset        iStartOffset,
                     TXNOffset       iEndOffset);
```

Set the current ranges font information. Values are passed in the attributes array. Values <= sizeof(UInt32) are passed by value. > sizeof(UInt32) are passed as a pointer. That is, the TXNTypeAttributes' 3rd field is a union that serves as either a 32-bit integer or a 32-bit pointer.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
iAttrCount: Count of type attributes in the TXNTypeAttributes array.
iAttributes[]: An array of attributes that application would like to set.
iStartOffset: The starting offset where the application would like to begin setting these attributes. If the goal is to change the current selection, the value of iStartOffset should be set to kTXNUseCurrentSelection (0xFFFFFFF).
iEndOffset: The offset where the style changes should stop. This is ignored if iStartOffset is equal to kTXNUseCurrentSelection

Output:
OSStatus: various MacOS errs. Notably memory manager and paramErrs.

```
EXTERN_API( OSStatus )
TXNSetTXNObjectControls(  TXNObject                    iTXNObject,
                          Boolean                      iClearAll,
                          ItemCount                    iControlCount,
                          TXNControlTag                iControlTags[],
                          TXNControlData               iControlData[]
                          );
```

Set things that apply to the entire TXNObject (i.e. the entire document). This includes line direction, justification, tab values, read-only status, whether the caret is on or off, whether the bottom-line window is used, text auto-wrap, keyboard synchronization, auto-indent, and application refcon. See the enum following the typedef for TXNControlTag for the list of constants that name what can be set. In addition, on systems which include ATSUI, all the ATSUI Line Control Attribute Tags can be passed to this function as a TXNControlTag. This is the case for all the ATSUI tags except kATSULineRotationTag. ATSUI Tags are applied to the entire TXNObject.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
iClearAll: reset all controls to the default
justification = LMTESysJust
line direction = GetSysDirection()
etc.
iControlCount: The number of TXNControlInfo records in the array.
iControlTags: An array[iControlCount] of TXNObject control tags.
iControlInfo: An array of TXNControlData structures which specify the type of information being set.

InputOutput:
OSStatus: paramErr or noErr.

```
EXTERN_API( OSStatus )
```

```
TXNGetTXNObjectControls(  TXNObject        iTXNObject,
                          ItemCount        iControlCount,
                          TXNControlTag    iControlTags[],
                          TXNControlData   oControlData[] );
```

Get the current TXNControls for the TXNObject. Specify tags in the iControlTags array. The values are returned in the oControlData array.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
iControlCount: The number of TXNControlInfo records in the array.
iControlTags: An array[iControlCount] of TXNObject control tags.

Input/Output:
OSStatus: paramErr or noErr.
oControlData: An array of TXNControlData structures which are filled out with the information that was requested via the iControlTags array. The application must allocate the array.

```
EXTERN_API( OSStatus )
TXNCountRunsInRange(TXNObject        iTXNObject,
                    UInt32           iStartOffset,
                    UInt32           iEndOffset,
                     ItemCount *     oRunCount);
```

Given a range specified by the starting and ending offset return a count of the runs in that range. Run in this case means changes in TextSyles or a graphic or sound.

Input:
iTXNObject The TXNObject you are interested in.
iStartOffset start of range
iEndOffset end of range
Output:
oRunCount count of runs in the range
OSStatus: paramerr

```
EXTERN_API( OSStatus )
TXNGetIndexedRunInfoFromRange(TXNObject                iTXNObject,
                              ItemCount                iIndex,
                              UInt32                   iStartOffset,
                              UInt32                   iEndOffset,
                              UInt32 *                 oRunStartOffset,
                              UInt32 *                 oRunEndOffset,
                              Collection *             oCollection) ;
```

Get information about the Nth run in a range. Should call TXNCountRunsInRange to get the count. The TXNTypeAttributes array must specify the type that the application is interested in. In other words, the tag field must be set. oTypeAttributes can be NULL.

Input:
iTXNObject Current TXNObject
iIndex the index is 0 based.
iStartOffset start of range
iEndOffset end of range
iTypeAttributeCount count of the number of TXNTypeAttribute strutures can be 0 if not interested in type attributes.
Output:
OSStatus paramErr or kRunIndexOutofBoundsErr.
oRunStartOffset start of run. This is relative to the beginning of the text, not the range
oRunEndOffset end of run.
oRunDataType Type of date contained in this run (i.e. PICT, moov, snd, TEXT) iTypeAttributeCount
oTypeAttributes Array of TXNTypeAttributes specifying the type attributes you are interested in.

```
EXTERN_API( ByteCount )
TXNDataSize (TXNObject               iTXNObject);
```

Return the size in bytes of the characters in a given TXNObject.
Input:
iTXNObject: The TXNObject
Output:
ByteCount: The bytes required to hold the characters

```
EXTERN_API( OSStatus )
TXNGetData(TXNObject           iTXNObject,
           TXNOffset           iStartOffset,
           TXNOffset           iEndOffset,
           Handle *            oDataHandle);
```

Copy the data in the range specified by startOffset and endOffset. This function should be used in conjunction with TXNNextDataRun. The application would call TXNNextDataRun to determine data runs and their size. For each data run of interest (i.e., one whose data the application wanted to look at), the application would call TXNGetData. The handle passed to TXNGetData should not be allocated.

TXNGetData takes care of allocating the dataHandle as necessary. However, the application is responsible for disposing the handle. No effort is made to ensure that data copies align on a word boundary. Data is simply copied as specified in the offsets.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject.
iStartOffset: absolute offset from which data copy should begin.
iEndOffset: absolute offset at which data copy should end.
Output:
OSStatus Memory errors or TXN_IllegalToCrossDataBoundaries if offsets specify a range that crosses a data type boundary.
oDataHandle: If noErr a new handle containing the requested data.

```
EXTERN_API( OSStatus )
TXNGetDataEncoded(TXNObject          iTXNObject,
                   TXNOffset         iStartOffset,
                   TXNOffset         iEndOffset,
                   Handle *          oDataHandle,
                   TXNDataType       encoding);
```

This function is similar to TXNGetData except for the following crucial difference. TXNGetDataEncoded only copies text. The application can specify whether text should be in the traditional Mac OS script encodings or Unicode. If the application specifies an encoding different from how the text is stored internally, the Text Encoding Conversion Manager will be invoked to translate the text into the requested encoding type.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject.
iStartOffset: absolute offset from which data copy should begin.
iEndOffset: absolute offset at which data copy should end.
encoding : should be kTXNTextData or kTXNUnicodeTextData
Output:
OSStatus Memory errors or TXN_IllegalToCrossDataBoundaries if offsets specify a range that crosses a data type boundary.
oDataHandle: If noErr a new handle containing the requested data.

```
EXTERN_API( OSStatus )
TXNSetDataFromFile (TXNObject        iTXNObject,
                     SInt16          iFileRefNum,
                     OSType          iFileType,
                     ByteCount       iFileLength,
                     TXNOffset       iStartOffset,
                     TXNOffset       iEndOffset);
```

Replace the specified range with the contents of the specified file. The data fork of the file must be opened by the application.

MLTE will not move the file's marker before reading the data. The marker must be set by the caller to the appropriate position before calling TXNSetDataFromFile. If the entire file is to be MLTE data then the marker should be set to position 0. If the caller wants to embed MLTE data within private or even other MLTE data then the file position must be set to the appropriate location.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
iFileRefNum: HFS file reference obtained when file is opened.
iFileType: files type.
iStartOffset: start position at which to insert the file into the document.
iEndOffset: end position of range being replaced by the file.
iFileLength Describes how much data should be read. This Parameter is ignored if the file type is thecustom file format that MLTE supports. This parameter is useful when a caller wishes MLTE to read data that is embedded in the callers private file. If you just want MLTE to deal with the whole file pass kTXNEndOffset (0x7FFFFFFF) for the iFileLength.
Output:
OSStatus: File manager error or noErr.

```
EXTERN_API( OSStatus )
TXNSetData (TXNObject                iTXNObject,
            TXNDataType              iDataType,
            void *                   iDataPtr,
            ByteCount                iDataSize,
            TXNOffset                iStartOffset,
            TXNOffset                iEndOffset);
```

Replace the specified range with the data pointed to by dataPtr and described by dataSize and dataType.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject.
iDataType: type of data must be one of TXNDataTypes.
iDataPtr: pointer to the new data.
iDataSize: Size of new data
iStartOffset: offset to beginning of range to replace
iEndOffset: offset to end of range to replace.
Output:
OSStatus: function result. parameter errors and Mac OS memory errors.

```
EXTERN_API( ItemCount )
TXNGetChangeCount(TXNObject          iTXNObject);
```

Retrieve the number of times document has been changed. The change count is incremented for every committed command. The count is cleared each time the TXNObject is saved. This function is useful for deciding if the Save item in the File menu should be active.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject

Output:
ItemCount: count of changes. This is total changes since document was created or last saved.

```
EXTERN_API( OSStatus )
TXNSave (TXNObject        iTXNObject,
         OSType           iType,
         OSType           iResType,
         TXNPermanentTextEncodingType  iPermanentEncoding,
         FSSpec*           iFileSpecification,
         SInt16            iDataReference,
         SInt16            iResourceReference );
```

Save the contents of the document as the type specified. The file to save the document to must be opened. If the file is being saved as plain text and the application has specified a resource type in which to save style attributes, then the resource fork of the file must be open as well.

The file marker of the opened file is expected to be at the position where the caller wants the data to be written. Typically, this is 0, but any valid file position can be used. MLTE does not move the marker before writing the file. This allows callers to write private data, followed by data that is written by MLTE which can subsequently be followed by more private data or even another MLTE file.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject.
iType: The file type to which the TXNObject should be saved. The type must be ‘txtn’, ‘TEXT’, or utxt.
iResType: The type of resource that should be used to save the style information if the file is being saved as plain TEXT. This parameter is ignored for other file types.
iPermanentEncoding: The encoding style in which to save the document. If the internal encoding being used by MLTE does not match the requested encoding type, the text is translated by the Text Encoding Conversion Manager.
iFileSpecification: A pointer to an FSSpec record that specifies the files location. This parameter is retained and used in calls to TXNRevert. It is not retained past the life of the TXNObject.
iDataReference: A reference to the files open data fork.
iDataReference: A reference to the files open resource fork. This parameter is ignored if the file type is not ‘TEXT’. You can save TEXT without style information by passing -1 for this parameter.

Output:
OSStatus: Function result. NoErr if document was saved. A File Manager error is returned if there was a failure.

```
EXTERN_API( OSStatus )
TXNRevert (TXNObject                iTXNObject);
```

Revert to the last saved version of this document. If the file was not previously saved, the document is reverted to an empty document.

TXNRevert does not support data embedding. To revert to data that is embedded in a private file type the caller should call TXNSetSelection to select all of the current data and then use TXNSetDataFromFile to read in the old data.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject
Output:
OSStatus: File manager errors, paramErr, or noErr.

```
EXTERN_API( OSStatus )
TXNPageSetup (TXNObject              iTXNObject);
```

Display the Page Setup dialog box for the current default printer and react to any changes (i.e., reformat the text if the page layout changes.)
Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject.
Output:
OSStatus: Print Manager errors, paramErr, noErr.

```
EXTERN_API( OSStatus )
TXNPrint (TXNObject                  iTXNObject);
```

Print the TXNObject formatted to fit the printer page size.
Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject.
Output:
OSStatus: Print Manager errors, paramErr, noErr.

```
EXTERN_API( Boolean )
TXNIsScrapPastable  (void);
```

Test to see if the current scrap contains data that is supported by MLTE. Used to determine if the Paste item in Edit menu should be active or inactive.
Output:
Boolean: function result. True if data type in Clipboard is supported. False if not a supported data type. If result is true, the Paste item in the menu should be highlighted.

```
EXTERN_API( OSStatus )
TXNConvertToPublicScrap (void);
```

Convert the MLTE private scrap to the public clipboard. This should be called on suspend events and before the application displays a dialog box that might support cut and paste. Or more generally, whenever someone other than MLTE needs access to the scrap data. The public formats supported are style text and styled Unicode text.

Output:

OSStatus: Function result. Memory Manager errors, Scrap Manager errors, noErr.

```
EXTERN_API( OSStatus )
TXNConvertFromPublicScrap (void);
```

Convert the public clipboard to MLTE private scrap . This should be called on resume events and after an application has modified the scrap.

Output:

OSStatus: Function result. Memory Manager errors, Scrap Manager errors, noErr.

```
EXTERN_API( void )
TXNGetViewRect (TXNObject            iTXNObject,
                Rect *               oViewRect);
```

Get the rectangle describing the current view into the document. The coordinates of this rectangle will be local to the window. If scroll bars are being managed by the TXNObject (i.e., the TXNNewObject flags include want vertical and horizontal scroll bars), the viewrect describes an area that encloses the scroll bars.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject.

Output:

oViewRect: The requested view rectangle.

```
EXTERN_API( OSStatus )
TXNFind (TXNObject                   iTXNObject,
        const TXNMatchTextRecord * iMatchTextDataPtr, /* can be NULL */
         TXNDataType                 iDataType,
         TXNMatchOptions             iMatchOptions,
         TXNOffset                   iStartSearchOffset,
         TXNOffset                   iEndSearchOffset,
         TXNFindUPP                  iFindProc,
         SInt32                      iRefCon,
         TXNOffset *                 oStartMatchOffset,
         TXNOffset *                 oEndMatchOffset);
```

Find a piece of text or a graphics object. Sounds are considered graphics objects in this context.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject.

iMatchTextDataPtr: ptr to a MatchTextRecord containing the text to match, the length of that text, and the TextEncoding the text is encoded in. This must be there if you are looking for text, but can be NULL if you are looking for a graphics object.

iDataType: the type of data to find. This can be any of the types defined in TXNDataType enum (TEXT, PICT, moov, snd). However, if PICT, moov, or snd is passed, then the default behavior is to match on any non-Text object. If you really want to find a specific type, you can provide a custom find callback or ignore matches that aren't the precise type you are interested in.

iStartSearchOffset: The offset at which a search should begin. The constant kTXNStartOffset specifies the start of the objects data.

iEndSearchOffset: The offset at which the search should end. The constant kTXNEndOffset specifies the end of the objects data.

iFindProc A custom callback. If will be called to match things rather than the default matching behavior.

iRefCon This can be use for whatever the application likes. It is passed to the FindProc (if a FindProc is provided.

Output:

oStartMatchOffset absolute offset to start of match. Set to 0xFFFFFFFF if there is no match.

oEndMatchOffset absolute offset to end of match. Set to 0xFFFFFFFF is no match. The default matching behavior is pretty simple for text: a basic binary compare is done. If the matchOptions say to ignore case, the characters to be searched are duplicated and case neutralized. This naturally can fail due to lack of memory if there is a large amount of text. It also slows things down. If MatchOptions say find an entire word, then once a match is found, an effort is made to determine if the match is a word. The default behavior is to test the character before and after to see if it is white space. If the kTXNUseEncodingWordRulesBit is set, than the Script Manager's FindWord function is called to make this determination. If the text being searched is Unicode text, then ATSUI’s word determining functions are used to determine the word. If the application is looking for a non-text type, then each non-text type in the document is returned. The FindProc is there to provide applications with more elaborate search engines (a regular expression processor, etc.) in mind.

```
EXTERN_API( OSStatus )
TXNSetFontDefaults (TXNObject        iTXNObject,
                    ItemCount        iCount,
                    TXNMacOSPreferredFontDescription  iFontDefaults[]);
```

For a given TXNObject, specify the font defaults for each script.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject.

iCount: count of FontDescriptions.

iFontDefaults: array of FontDescriptins.

Output:

OSStatus: function result (memory error, paramErr)

```
EXTERN_API( OSStatus )
```

```
TXNGetFontDefaults (TXNObject        iTXNObject,
                    ItemCount *      ioCount,
                    TXNMacOSPreferredFontDescription  iFontDefaults[]);
```

For a given TXNObject, make a copy of the font defaults.

Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject.
iCount: count of FontDescriptions in the array.
iFontDefaults: array of FontDescriptins to be filled out.
Output:
OSStatus: function result (memory error, paramErr). To determine how many font descriptions need to be in the array, you should call this function with a NULL for the array. iCount will return with the number of font defaults currently stored.

```
EXTERN_API( OSStatus )
TXNAttachObjectToWindow (TXNObject        iTXNObject,
                         GWorldPtr        iWindow,
                         Boolean          iIsActualWindow);
```

If a TXNObject was initialized with a NULL window pointer, use this function to attach a window to that object. In version 1.0 of MLTE, attaching a TXNObject to more than one window is not supported.
Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject.
iWindow: GWorldPtr that the object should be attached to
iIsActualWindow: True if the GWorldPtr was obtained by calling NewWindow or NewCWindow. False if it is a generic port. Passing false means that MLTE will never call window-specific Toolbox functions like InvalRect, BeginUpdate, etc. If false is passed, it is the application's responsibilty to handle this type of functionality if it is required.
Output:
OSStatus: function result (kObjectAlreadyAttachedToWindowErr, paramErr)

```
EXTERN_API( Boolean )
TXNIsObjectAttachedToWindow (TXNObject   iTXNObject);
```

A utility function that allows a application to check a TXNObject to see if it is attached to a window.
Input:
iTXNObject: opaque TXNObject obtained from TXNNewObject.
Output:
Boolean: function result. True if object is attached. False if TXNObject is not attached.

```
EXTERN_API( OSErr )
TXNDragTracker (TXNObject            iTXNObject,
                TXNFrameID           iTXNFrameID,
                DragTrackingMessage  iMessage,
                WindowPtr            iWindow,
                DragReference        iDragReference,
                Boolean              iDifferentObjectSameWindow );
```

If you ask that drag-handling procs not be installed by passing kTXNDoNotInstallDragProcsMask to TXNNewObject, you should call this function when your drag tracker is called and you want MLTE to take over.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject.
iTXNFrameID   TXNFrameID obtained from TXNNewObject
iMessage   drag message obtained from Drag Manager
iWindow   windowPtr obtained from Drag Manager
iDragReference   dragReference obtained from Drag Manager
iDifferentObjectSameWindow: If your application is displaying more than one TXNObject per window, pass true here when the drag operation moves out of one object's view rectangle and into another TXNObject's view rectangle.

Output:

OSErr: function result. OSErr is used over OSStatus so that it matches the Drag Manager definition of Tracking callback

```
EXTERN_API( OSErr )
TXNDragReceiver (TXNObject           iTXNObject,
                 TXNFrameID          iTXNFrameID,
                 WindowPtr           iWindow,
                 DragReference       iDragReference,
                 Boolean             iDifferentObjectSameWindow);
```

If you are handling Drag and Drop (i.e., you passed kTXNDoNotInstallDragProcsMask to TXNNewObject), call this when your drag receiver is called and you want MLTE to take over.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject.
iTXNFrameID TXNFrameID obtained from TXNNewObject
iWindow   windowPtr obtained from Drag Manager
iDragReferencedragReference obtained from Drag Manager

Output:

OSErr: function result. OSErr is used over OSStatus so that it matches the Drag Manager definition of Tracking callback

```
EXTERN_API( OSStatus )
TXNActivate (TXNObject                    iTXNObject,
             TXNFrameID                   iTXNFrameID,
             TXNScrollBarState            iActiveState);
```

Make the TXNObject active in the sense that it can be scrolled if it has scroll bars. If the TXNScrollBarState parameter is true, then the scroll bars will be active even when the TXNObject is not focused (i.e., the insertion point is not active)

This function should be used if you have multiple TXNObjects in a window, and you want them all to be scrollable even though only one at a time can have the keyboard focus.

Input:

iTXNObject: opaque TXNObject obtained from TXNNewObject.

iTXNFrameID   TXNFrameID obtained from TXNNewObject

iActiveState   Boolean. If true, scroll bars stay active even though TXNObject does not have the keyboard focus.   If this parameter is false, scroll bars are synced with active state (i.e., a focused object has an active insertion point or selection and active scroll bars. An unfocused object has inactive selection—grayed or framed selection—and inactive scroll bars.) The latter state is the default and usually the one you use if you have one TXNObject in a window.

Output:

OSStatus: function result. ParamErr if bad iTXNObject or frame ID.

```
EXTERN_API( OSStatus )
TXNSetBackground ( TXNObject          iTXNObject,
                   TXNBackground *    iBackgroundInfo);
```

Set the type of background the TXNObject's text, etc., is drawn onto. The background can be a color or a picture.

Input:

iTXNObject: opaque TXNObject obtained from IncomingDataFilter callback.

iBackgroundInfo: struct containing information that describes the background

Output:

OSStatus: function result. paramErrs.

```
EXTERN_API( OSStatus )
TXNNewFontMenuObject(   MenuHandle            iFontMenuHandle,
                        SInt16                iFontMenuID,
                        SInt16                iStartHierMenuID,
                        TXNFontMenuObject*    oTXNFontMenuObject );
```

Get a new TXNFontMenuObject. A TXNFontMenuObject is an obaque structure that describes and handles all aspects of user interaction with a Font menu. The menu is created dynamically. The application provides the menu title, the menu ID, and the menu ID to use if any hierarchical menus are created. Hierarchical menus are created on systems with ATSUI.

Input:

IFontMenuHandle An empty menu handle (well the title is there) that the caller created via NewMenu or GetNewMenu. This menu handle should not be disposed before the returned TXNFontMenuObject has been disposed via TXNDisposeFontMenuObject /:

.

iFontMenuID: The menu ID that the font menu should have.
iStartHierMenuID: The menu ID at which hierarchical menu IDs will begin.

Output:
OSStatus: function result, Memory Error, paramError.
oTXNFontMenuObject: A new TXNFontMenuObject is returned.

```
EXTERN_API( OSStatus )
TXNGetFontMenuHandle(   TXNFontMenuObject     iTXNFontMenuObject,
                        MenuHandle*           oFontMenuHandle );
```

Get the Font menu handle that belongs to a TXNFontMenuObject.

Input:
oTXNFontMenuObject: TXNFontMenuObject obtained from TXNNewFontMenuObject.

Output:
OSStatus: function result, ParamError.
oFontMenuHandle: The Font menu created when TXNNewFontMenuObject was created. The application should NOT dispose of this Handle.

```
EXTERN_API( OSStatus )
TXNDoFontMenuSelection(  TXNObject                 iTXNObject,
                         TXNFontMenuObject         iTXNFontMenuObject,
                         SInt16                    iMenuID,
                         SInt16                    iMenuItem );
```

Pass the results of MenuSelect to this routine. If the iMenuID is the Font menu or one of its sub-menus, the currently selected text will be changed to the font the user selected.

Input:
iTXNObject: TXNObject obtained from TXNNewObject;
iTXNFontMenuObject: TXNFontMenuObject obtained from TXNNewFontMenuObject.
iMenuID: The high 16-bits of the long word returned by MenuSelect. It is necessary to pass the menuID because the font menu may have hierarchical sub-menus.
iMenuItem: The low 16-bits of the result of MenuSelect.

Output:
OSStatus: function result, ParamError.

```
EXTERN_API( OSStatus )
TXNPrepareFontMenu(    TXNObject                 iTXNObject,
                       TXNFontMenuObject         iTXNFontMenuObject );
```

Prepare a Font menu for display. If the TXNObject’s current selection is a single font, the item for that font is checked. If iTXNObject is NULL, the menu is grayed out.

Input:
iTXNObject: TXNObject obtained from TXNNewObject;
iTXNFontMenuObject: TXNFontMenuObject obtained from TXNNewFontMenuObject.

Output:
OSStatus: function result, ParamError.

```
EXTERN_API( OSStatus )
TXNDisposeFontMenuObject(TXNFontMenuObject          iTXNFontMenuObject );
```

Dispose a Font menu object. This function calls DisposeMenuHandle on the Font menu handle.

Input:
iTXNFontMenuObject: TXNFontMenuObject obtained from TXNNewFontMenuObject.

Output:
OSStatus: function result, ParamError.

```
EXTERN_API_C( OSStatus )
TXNEchoMode (TXNObject              iTXNObject,
             UniChar                iechoCharacter,
             TextEncoding           iencoding,
             Boolean                ion);
```

Put the TXNObject into echo mode. When a TXNObject is in echo mode all characters in the TXNObject have the character specified by 'echoCharacter' substituted for the actual glyph when drawing occurs. Note that the echoCharacter is typed as a UniChar, but this is done merely to facilitate passing any two byte character. The encoding parameter actually determines the encoding used to locate a font and display a character. Thus if you wanted to display the diamond found in the Shift-JIS encoding for MacOS you would pass in 0x86A6 for the character but an encoding that was built to represent the MacOS Japanese encoding.

Input:
iTXNObject: opaque TXNObject obtained from IncomingDataFilter callback.
iechoCharacter: character to use in substitution
iencoding: encoding from which character is drawn.
ion: TRUE if turning EchoMode on. False if turning it off.

Output:
OSStatus: function result. paramErrs.

```
EXTERN_API( OSStatus )
TXNVersionValue TXNVersionInformation( TXNFeatureBits* oFeatureFlags );
```

Get the version number and a set of feature bits. The initial version number is And the only bit used in the oFeatureFlags is the lsb: 0x00000001

```
      Input:
            NONE

      Output:
         TXNVersionValue:  Current version.
         TXNFeatureBits*:  Pointer to a bit mask.  See TXNFeatureMask enum
                           above. If kTXNWillDefaultToATSUIBit is set it
                           means that by default MLTE will use ATSUI to image
                           and measure text and will default to using Unicode
                           to store characters.
```