New Technical Notes

Macintosh

Developer Support

TE 27 – Inline Input for TextEdit with TSMTE

Text

Revised by: Norbert Lindenberg March 1994
Written by: Yasuo Kida, Keisuke Hara, Nobuhiro Miyatake, December 1993
Peter Sparks, Norbert Lindenberg

This Technical Note describes TSMTE, an extension that simplifies implementation of inline input for TextEdit using the Text Services Manager in System 7.1 and later, and shows you how to make the best use of it. It also contains some advice for working with the Text Services Manager that applies to any application using TSM, not just those using TSMTE.
Changes since December 1993: Fixed a bug in the IntlTSMEvent routine that could modify the Window Manager port's font, and updated the related information about the Japanese input method. Added section "Input Methods Need Null Events". Clarified the information about required support for Apple events in the Prerequisites section.

Topics

- What is TSMTE
- **Where and how is TSMTE delivered**
- **How is TSMTE used in an application**
- **Advice for working with the Text Services Manager**

Introduction

System 7.1 introduced a new Toolbox manager, the Text Services Manager, that makes it easier for applications to provide inline input for 2-byte script systems (see *Inside Macintosh: Text*, pages 7-6 to 7-12, for an overview). To implement inline input using the Text Services Manager, an application has to do two things: make some calls to the Text Services Manager in the right places, and provide Apple event handlers that respond to events from input methods by updating text data structures, displaying text, and translating text offsets and screen coordinates. The first part is easy (maybe 50-100 lines of code), the second part can, depending on the complexity of the text engine that you use, require substantial work (several hundred to thousands of lines of code).

TSMTE is an extension to the Text Services Manager that does the second part of the work for you if you use TextEdit. It provides Apple event handlers that handle all interactions between an input method and TextEdit. The handlers are kept in the system heap, so they are shared between all applications.

TSMTE thus can reduce the effort needed to implement inline input to a day or two. If you use TextEdit in documents or modeless dialogs, you have to make a few calls to TSMTE in exchange for its help; if you use TextEdit only in modal dialogs, you only have to modify the DLOG resources that define them.

If you use a text engine other than TextEdit for editing in your application, you can still use TSMTE to handle inline input wherever you use TextEdit, e.g., in dialogs.

Note: KanjiTalk 6.0.7 had an extension that provided fully automatic support for inline input with TextEdit without any kind of modification to applications. With TSMTE, inline input is not quite so automatic – you have to make the necessary calls to TSMTE and the Text Services Manager to make it work.

Note: Inside Macintosh: Text, pages 2-107 to 2-109, discusses two feature bits, teFUseTextServices and teFInlineInput. TeFUseTextServices doesn't have any impact on the Text Services Manager or TSMTE, so you can safely ignore it. TeFInlineInput is handled by TSMTE, so there is no need for you to touch it.

TSMTE Overview

Availability

Currently, TSMTE is delivered as a system extension called "Inline Tuika Kinou", which is shown here:

Figure 1–Inline Tuika Kinou extension file

If you look for this file while running system software in some language using the Roman script system, the name of the extension file will be displayed as "ÉCÉìÉâÉCÉìí«â¡ã@ñ\".

The extension is part of the Japanese version of System 7.1, KanjiTalk 7. It is not part of the Japanese Language Kit or of any other version of System 7.1. This means that if you use TSMTE to implement inline input, it will currently only benefit users of Japanese system software. However, TSMTE may get rolled into a future version of the base system software, so that it would then be available for use with any 2-byte script system on any Macintosh worldwide. Your application should therefore use Gestalt to check for the presence of TSMTE and use it whenever it is available.

For development purposes, if you don't feel comfortable using Japanese system software, you can install the Japanese Language Kit and the Inline Tuika Kinou extension into system software in any language that you like. Apple does not exactly guarantee that these configurations will work, but our experience so far has been good. Final testing should of course be done on a real Japanese system. In order to get your application to use a Japanese font in dialogs or in documents where you don't allow the user to select a font, you have to register your application as Japanese using the Language Register application that comes with the Japanese Language Kit.

Limitations

You may find that with some fonts TSMTE truncates the lower portion of characters drawn in the input area because it reserves a two-pixel high area for underlining. This problem will be fixed in a future release.

Prerequisites

To use the Text Services Manager and TSMTE, your application has to support Apple events. At a minimum, it has to install at least one Apple event handler using AEInstallEventHandler. Usually of course you would install handlers for the four required event types, and set the isHighLevelEventAware bit in the SIZE -1 resource.

Note: The SIZE resource also has a useTextEditServices bit. Contrary to the comments in Types.r and in *Inside Macintosh: Text*, this bit doesn't have any influence on inline input done with the Text Services Manager and TSMTE. It was used by the extension that provided inline input for TextEdit in KanjiTalk 6.0.7.

Preparing to Use TSMTE

Starting from this section, we assume that you have a good understanding of the high-level routines of the Text Services Manager. You can find the necessary information in *Inside Macintosh: Text*, pages 7-17 to 7-24. Please make sure to read the section "More Inline Input Advice" at the end of this Technical Note for additional hints on how to successfully use the Text Services Manager and input methods.

Defining the Level of Functionality You Need

Your application can provide several different kinds of text editing functionality, and your usage of TSMTE and the Text Services Manager depends on which one you offer. Your application may support text input:
1) in modal dialogs (which use TextEdit)
2) in document windows using TextEdit and in modeless dialogs
3) in windows using your own text engine
plus in any combination thereof.

The rest of this section discusses the steps you have to take to prepare your application to use TSMTE in any possible context. The following sections then look at the individual cases.

Testing for TSMTE

Before making calls for TSMTE, you have to check for its presence using Gestalt. The selector for TSMTE is gestaltTSMTEAttr, and you know that it is available if the gestaltTSMTEPresent bit in the response is set.

The following code initializes separate Boolean variables to indicate the presence of the Text Services Manager and TSMTE. Having separate variables is useful if you use your own text engine in addition to TextEdit and want to support inline input for that engine even if TSMTE is not present.

```
static void CheckForTextServices(void)
{
   long gestaltResponse;

   gHasTextServices = false;            // unless proven otherwise
   gHasTSMTE = false;                   // unless proven otherwise

   if (TrapAvailable(_Gestalt))
   {
      if ((Gestalt(gestaltTSMgrVersion, &gestaltResponse) == noErr) &&
          (gestaltResponse >= 1))
      {
         gHasTextServices = true;
         if (Gestalt(gestaltTSMTEAttr, &gestaltResponse) == noErr)
            gHasTSMTE = BTst(gestaltResponse, gestaltTSMTEPresent);
      };
   };
}
```

Initializing and Closing the Text Services Manager

To enable inline input, you have to initialize the Text Services Manager by calling InitTSMAwareApplication in your initialization sequence, and close it before quitting by calling CloseTSMAwareApplication. But what if TSMTE is not available? Then you have to make the floating input window available to the user for entering text in a 2-byte script into a TextEdit field. How to do this depends on whether you use your own text engine in addition to TextEdit (we assume that your own text engine supports inline input – otherwise you probably would not think about adding inline input support to TextEdit).

If you only use TextEdit in your application, you simply call InitTSMAwareApplication only if TSMTE is available. If you don't call InitTSMAwareApplication, system software will automatically handle input in 2-byte scripts for your application in a floating input window. The following code performs the initialization and also shows how to move on without text services if InitTSMAwareApplication fails:

```
if (!(gHasTSMTE && InitTSMAwareApplication() == noErr))
{
   // if this happens, just move on without text services
   gHasTextServices = false;
   gHasTSMTE = false;
};
```

If you use your own text engine in addition to TextEdit and support inline input for it, you want to use inline input for your engine even if TSMTE is not available. To do this, you simply check gHasTextServices instead of gHasTSMTE in the first line of the code above. Later, you also have to make sure that you tell the Text Services Manager to use the floating input window whenever a TextEdit field is active – the section "Using TSMTE and TextEdit in Addition to Your Own Text Engine" discusses this in detail.

In either case, you have to call CloseTSMAwareApplication before quitting an application for which you successfully called InitTSMAwareApplication:

```
if (gHasTextServices)
   (void) CloseTSMAwareApplication();
ExitToShell();
```

Using TSMTE for Modal Dialogs

Once you have initialized the Text Services Manager, TSMTE offers a very easy way to handle inline input for modal dialogs: you set the refCon field in the DLOG resource to kTSMTEDialog or kTSMTEInterfaceType, TSMTE handles the rest. Or, if circumstances force you to create a dialog programmatically, you can pass kTSMTEInterfaceType as the refCon argument to NewDialog or NewCDialog (these routines do not accept kTSMTEDialog). Either way, TSMTE will automatically create a TSMDocument for you, activate and deactivate the TSMDocument, and enable inline input for the dialog. When you call CloseDialog or DisposeDialog, TSMTE disposes of the TSMDocument it created, activates the TSMDocument that was active before opening the dialog (if there was one), and resets the Text Services Manager flag that determines whether to use the floating input window to its previous state.

Note: If you use this feature, you can still use the refCon field for your own purposes after creating the dialog – TSMTE doesn't need it any longer.

The difference between the two constants is that kTSMTEInterfaceType tells TSMTE to use an extended dialog record, TSMDialogRecord, while kTSMTEDialog uses the standard dialog record. Using the extended dialog record lets you access the information that TSMTE uses if you need it; without it, the information is stored in TSMTE's private data structures. If you use kTSMTEInterfaceType but don't provide storage for the dialog record, the Dialog Manager routines will automatically allocate an extended record.

If you use kTSMTEInterfaceType and allocate your own storage or add your own fields to the dialog record, you have to take the additional 20 bytes of the extended dialog record into account. If you add your own fields, it's a good idea to also allocate your own storage. This way you can always include the fields for TSMTE, otherwise the location of your fields in the record depends on whether TSMTE is installed or not.

If you provide an event filter function for ModalDialog, TSMTE gives you the choice whether you want to handle Text Services Manager calls or whether TSMTE should do it. To determine whether the function handles the calls, TSMTE calls it with a null event. If the function calls TSMEvent, TSMTE assumes that the function makes all the necessary calls to the Text Services Manager. If it doesn't, or if there is no event filter function, TSMTE makes all the necessary calls itself.

While using inline input, ModalDialog doesn't return because it doesn't get to see any "real" events. If you have a dialog that opens with a disabled action button and waits for the user to type text into an editable text item before enabling the button, checking the editable text item only after ModalDialog returns does not have the desired effect – if the user uses inline input to enter text, the button doesn't get enabled. A solution for this is to use an event filter function that checks the text in the editable text item and enables the button if the text length is above zero. The event filter function is guaranteed to be called with a null event whenever TSMEvent consumes a keyDown event.

Using TSMTE for Document Windows and for Modeless Dialogs

If you use TextEdit to edit text in document windows or if you use modeless dialogs, some more work is shifted over to you: now it becomes your responsibility to call the high-level Text Services Manager routines. You'll have to add calls to:

```
NewTSMDocument
DeleteTSMDocument
ActivateTSMDocument
DeactivateTSMDocument
TSMEvent
TSMMenuSelect
SetTSMCursor
FixTSMDocument
```

Before making the Text Services Manager calls, you have to make sure that their preconditions are met. You only want to create a TSM document for a TextEdit text record if TSMTE is available. You only want to delete, activate, or deactivate, or confirm ("fix") a TSM document if creating it was successful. The remaining routines depend on the Text Services Manager being available, but not necessarily on TSMTE – you may be using your own text engine with inline input in addition to TextEdit. The variables gHasTextServices and gHasTSMTE introduced above can help you make the necessary decisions. For example, your menu handling code might look like this:

```
menuResult = MenuSelect(event->where);
if (!(gHasTextServices && TSMMenuSelect(menuResult)))
   DoMenuCommand(menuResult);
HiliteMenu(0);
```

The usage of Text Services Manager routines is documented in *Inside Macintosh: Text*, so we'll discuss only how TSMTE extends the Text Services Manager interface.

Creating a TSM Document

When creating a TSMDocument for a TextEdit text record, you have to use a special interface type kTSMTEInterfaceType to indicate that TSMTE should handle Apple events for this TSM document.

If you pass kTSMTEInterfaceType to NewTSMDocument, the refCon argument takes on a different meaning. Instead of a value to be stored in the TSM document, you should pass in the address of a variable of type TSMTERecHandle. TSMTE allocates a data structure of type TSMTERec and assigns a handle to it to your variable. This data structure contains several fields that you can use to tailor TSMTE's behavior to the needs of your application. It is your application's responsibility to initialize the record.

```
struct TSMTERec {
   TEHandle            textH;
   TSMTEPreUpdateUPP                   preUpdateProc;
   TSMTEPostUpdateUPP  postUpdateProc;
   long                updateFlag;
   long                refCon;
};
```

The textH field has to be set to the text record handle that this TSM document relates to.

In preUpdateProc and postUpdateProc you can specify call-back routines that TSMTE should call before and after its own code when handling the Update Active Input Area event (one of the Apple events that is sent by the input method). The interfaces and possible uses for both routines are described below. If you don't have routines that TSMTE should call, set the fields to nil.

The updateFlag field is intended for customization of TSMTE's behavior. The idea is that TSMTE can define several constants for variations in its behaviors, and you sum up the constants for the variations that you like and assign them to the updateFlag field. Currently, only one such constant, kTSMTEAutoScroll, is defined. It specifies that TSMTE automatically scrolls the selection range into view. If you set updateFlag to 0, automatic scrolling is disabled, and you have to scroll the text yourself, e.g., in one of the call-back routines.

The refCon field lets you specify a value that TSMTE will pass on to the call-back routines. TSMTE doesn't make any other use of this field.

Here is some sample code for creating a TSMDocument. It assumes that you have just created a TEHandle called docTEHandle, and that a Boolean variable good is used to indicate whether operations are successful, and that you want to pass a pointer to the document window to your call-back routine.

```
if (good && gHasTSMTE)
{
   supportedInterfaces[0] = kTSMTEInterfaceType;
   if (NewTSMDocument(1, supportedInterfaces, &doc->docTSMDoc,
            (long) &doc->docTSMTERecHandle) == noErr)
   {
      TSMTERecPtr tsmteRecPtr = *(doc->docTSMTERecHandle);

      tsmteRecPtr->textH = doc->docTE;
      tsmteRecPtr->preUpdateProc = gTSMTEPreUpdateUPP;
      tsmteRecPtr->postUpdateProc = gTSMTEPostUpdateUPP;
      tsmteRecPtr->updateFlag = kTSMTEAutoScroll;
      tsmteRecPtr->refCon = (long) window;
   }
   else
      good = false;
};
```

You shouldn't dispose of the TSMTERecHandle – DeleteTSMDocument will do this for you.

Using a Pre-Update Call-Back Routine

Pre-update call-back routines for TSMTE have the following interface:

```
pascal void MyTSMTEPreUpdateProc(TEHandle textH, long refCon);
```

If you provide a pre-update routine for a TSM document, it is called before TSMTE's code for handling the Update Active Input Area events relating to this document.

The values for the textH and refCon arguments are taken from the TSMTERecHandle of the TSM document.

One common use of the pre-update routine is to save information that will be needed for Undo and Redo. Without inline input, an application typically treats an uninterrupted sequence of keyDown events (other than arrow or function keys) as one action, and saves the currently selected text and related information when receiving the first event in this sequence. In this regard, you should treat a call to your pre-update routine as just another form of typing, and if it's the first one in a typing sequence, save the information for Undo.

Another use is to work around a bug in TSMTE 1.0, which doesn't always synchronize the font to be used with the current keyboard script. The following routine checks whether the current font can display the incoming characters, and if not, sets the font to the keyboard script's application font. A better solution would be to scan the text backwards for the most recently used font of the keyboard script. This solution will be used by future versions of TSMTE, so make sure to check the TSMTE version and use the workaround only for TSMTE 1.0, as shown below. The synchronization is only necessary when a new active input area is created, so you may want to use the post-update routine to track whether there is an active input area and only execute the font synchronization code when the pre-update routine is called while there's no active input area.

```
static pascal void MyTSMTEPreUpdateProc(TEHandle textH, long refCon)
{
   long response;
   ScriptCode keyboardScript;
   short mode;
   TextStyle theStyle;

   if ((Gestalt(gestaltTSMTEVersion, &response) == noErr) &&
       (response == gestaltTSMTE1))
   {
      keyboardScript = GetScriptManagerVariable(smKeyScript);
      mode = doFont;
      if (!(TEContinuousStyle(&mode, &theStyle, textH) &&
            FontToScript(theStyle.tsFont) == keyboardScript))
      {
         theStyle.tsFont = GetScriptVariable(keyboardScript, smScriptAppFond);
         TESetStyle(doFont, &theStyle, false, textH);
      };
   };
}
```

Note: Depending on which interface files you use, you may have to use the old name GetEnvirons instead of the new GetScriptManagerVariable, because there's no correct declaration for the new name. The universal interfaces have a correct declaration.

If your application occasionally changes the origin of the TextEdit record's grafPort, you can also use the pre-update routine to reset the origin so that characters get drawn in the right location.

Using a Post-Update Call-Back Routine

Post-update call-back routines for TSMTE have the following interface:

```
pascal void MyTSMTEPostUpdateProc(TEHandle textH, long fixLen,
        long inputAreaStart, long inputAreaEnd,
        long pinStart, long pinEnd, long refCon);
```

If you provide a post-update routine for a TSM document, it is called after TSMTE's code for handling Update Active Input Area events relating to this document. If you have set the updateFlag field in the TSMTERec record to kTSMTEAutoScroll, TSMTE calls the call-back routine first, and then scrolls the selection range into view.

The values for the textH and refCon arguments are taken from the TSMTERecHandle of the TSM document. InputAreaStart and inputAreaEnd are the offsets of the start and end of the

active input area relative to the entire text handle; they are both set to -1 if there is no active input area. The remaining parameters are a subset of the parameters for the Update Active Input Area event. The fixLen parameter is the length of the confirmed text. PinStart and pinEnd are the offsets of the start and end of the text range that should be in view.

Common uses of the post-update routine are:
- adjusting scroll bars or input field widths to the width and height of the text, which may have changed during editing,
- setting a "modified" flag for the document,
- saving information about the text being entered for Undo and Redo,
- keeping track of whether there's an active input area.

Calling FixTSMDocument

The FixTSMDocument routine should be called whenever the user switches from typing to a different kind of activity that operates on the text, e.g., initiating an editing command from the menu or selecting text. It should not be called for actions that would not be considered interrupting a typing sequence, e.g., resizing or scrolling the window. There are some actions in between for which we don't have clear guidelines yet; in these cases use your best judgment.

TSMTE will in some cases detect that FixTSMDocument needs to be called and do it for you, e.g., when the user clicks into a part of the document outside the input area. In most cases however it is your responsibility to call FixTSMDocument when appropriate: when the user selects an editing command from the menu, closes a document, saves or prints it.

Using TSMTE and TextEdit in Addition to Your Own Text Engine

If you use your own text engine in addition to TextEdit, you will have to provide your own Apple event handlers for the Text Services Apple events to implement inline input for your engine. However, you can still use TSMTE to provide inline input wherever you use TextEdit in your application.

Your Apple event handlers don't need to worry about TextEdit at all. TSMTE installs its event handlers in the system heap, so you can install your handlers in the application heap. The supported interface type that you specify when you create a TSM document is used to arbitrate between the handlers: for TSM documents that were created with kTSMTEInterfaceType, the TSMTE handlers are called, for those that were created with kTextService, your handlers.

The only thing you have to worry about is what to do if the Text Services Manager is available, but not TSMTE, so that you can provide inline input for your engine, but not for TextEdit. In this case you want to make sure that inline input is used whenever your engine is active, but that a floating input window is made available whenever a TextEdit field is active (without the floating input window users would not be able to type anything meaningful in a 2-byte script). You can do this by calling UseInputWindow whenever you activate or deactivate a TextEdit record – here is sample code for activation:

```
if (doc->docTSMDoc != nil)
   CheckError(ActivateTSMDocument(doc->docTSMDoc));
else
   CheckError(UseInputWindow(nil, true));
```

More Inline Input Advice

This section contains some information that is not specific to TSMTE, but applies to all applications that use the Text Services Manager in any form. It shows workarounds for some unexpected features (we won't use entomological terminology here…) in the Text Services Manager and first-generation input methods written for it. Some of those unexpected features are expected to be or have already been discontinued in newer versions.

Incorrect Declaration in TextServices.p

The Text Services interface file TextServices.p that is currently distributed on E.T.O. 12 and the November 1993 Developer CD contains an incorrect declaration for NewTSMDocument. The declaration should read:

```
Function NewTSMDocument(numOfInterface: Integer;
      VAR supportedInterfaceTypes: InterfaceTypeList; VAR idocID: TSMDocumentID;
   refCon: Longint): OSErr;
   INLINE $303C, $0000, $AA54;
```

Without the keyword "var" in front of "supportedInterfaceTypes", your application will encounter a bus error in NewTSMDocument. So, go in and add the keyword "var" if it's not there.

The declaration for NewTSMDocument in TextServices.h is correct.

DeleteTSMDocument Uses Disposed Handle

If a TSMDocument is deleted without being deactivated first, the routine DeleteTSMDocument may reuse a handle that it has already disposed of. It dereferences this handle and writes a single byte. This may eventually cause your application to crash mysteriously.

Workaround: make sure to deactivate each TSM document using DeactivateTSMDocument before calling DeleteTSMDocument to delete it. If TSMTE calls DeleteTSMDocument for a TSM document it created for a modal dialog it does the right thing.

ActivateTSMDocument Must Be Called From Foreground

ActivateTSMDocument does not work properly if called from the background. When a window that owns a TSM document is coming to the foreground from the background, your application is supposed to call ActivateTSMDocument with the TSMDocumentID for that window. However, if your application makes this call while still in the background, inline input may not become re-enabled in that window. Applications that follow the usual scheme of activating windows after receiving an activate event shouldn't have any problems.

SetTSMCursor and Cursor Regions

Applications that use TSM are supposed to call SetTSMCursor generously in order to allow TSM components to set the cursor when they need to do so. Unfortunately, there is no protocol for finding out in which region the input method would want to set the cursor. This presents a problem for many applications that try to be cooperative by passing a cursor region to WaitNextEvent. Without information from the input method, an application using inline input cannot calculate a meaningful cursor region, and thus has to be run whenever the mouse moved.

The only thing you can currently do is to define a 1-pixel cursor region under the mouse point and pass this region to WaitNextEvent. This setup will cause mouse-moved events to be generated whenever the mouse is moved, but will let the application sleep if the mouse stays put.

Strange Delete Key Behavior

If you use version 1.0 of Kotoeri, Apple's Japanese input method, you may notice that deletion of Japanese characters in the active input area using the delete character does not work properly in your application. This is more likely to happen with styled text or on non-Japanese system software (e.g., with the Japanese Language Kit installed). Kotoeri thinks that it is using a Roman font and only deletes one byte for each delete key pressed. This results in the need to press the delete key twice to delete a single Japanese character as well as other strange inline behavior. This bug has been fixed in version 1.1.1, which was first shipped as part of the Japanese version of the system software for Power Macintosh.

Workaround: Add some code around your call to TSMEvent that sets the font in the current grafPort to one of the keyboard script, and resets it afterwards if necessary. Kotoeri's behavior depends on the font in the current grafPort that it encounters during your call to TSMEvent. Here is a routine that you can call instead of TSMEvent to accomplish this:

```
static Boolean IntlTSMEvent(EventRecord *event)
{
   short oldFont;
   ScriptCode keyboardScript;

   // make sure we have a port and it's not the Window Manager port
   if (qd.thePort != nil && FrontWindow() != nil)
   {
      oldFont = qd.thePort->txFont;
      keyboardScript = GetScriptManagerVariable(smKeyScript);
      if (FontToScript(oldFont) != keyboardScript)
         TextFont(GetScriptVariable(keyboardScript, smScriptAppFond));
   };
   return TSMEvent(event);
}
```

You should also make sure that the current grafPort at this point is the one in which the input will be displayed.

Input Methods Need Null Events

Some input methods rely on receiving null events to function properly. The effects you may see if you don't feed them enough null events vary: An input method may not redraw its windows correctly, or it may occasionally lose input data.

Workaround: you should call IntlTSMEvent (or TSMEvent if you don't use the workaround in the previous section) even if WaitNextEvent returns false. The sample code in *Inside Macintosh: Text ,* page 7-22, is wrong in this respect. Your code might look like this:

```
gotEvent = WaitNextEvent(everyEvent, &event, GetSleep(), cursorRgn);
if (gHasTextServices && (gotEvent || event.what == nullEvent))
   if (IntlTSMEvent(&event))
      gotEvent = false;
if (gotEvent)
```

```
      HandleEvent(&event);
   else
      DoIdle();
```

Pascal Summary

Constants

```
   const

      { signature, interface types }

      kTSMTESignature = 'tmTE';
      kTSMTEInterfaceType = kTSMTESignature;
      kTSMTEDialog = 'tmDI';

      { Gestalt }

      gestaltTSMTEAttr = kTSMTESignature;
      gestaltTSMTEPresent = 0;
      gestaltTSMTEVersion = 'tmTV';
      gestaltTSMTE1 = $0100;

      { update flag for TSMTERec }

      kTSMTEAutoScroll = 1;
```

Data Types

```
   type

      TSMTERec = record
         textH:           TEHandle;
         preUpdateProc:                    ProcPtr;
         postUpdateProc:  ProcPtr;
         updateFlag:      Longint;
         refCon:          Longint;
         end;

      TSMTERecPtr = ^TSMTERec;
      TSMTERecHandle = ^TSMTERecPtr;

      TSMDialogRecord = record
         fDialog:         DialogRecord;
         fDocID:          TSMDocumentID;
         fTSMTERecH:      TSMTERecHandle;
         fTSMTERsvd:      array [0..2] of Longint;
         end;

      TSMDialogPeek = ^TSMDialogRecord;
```

Application-Defined Routines

```
   procedure MyTSMTEPreUpdateProc(textH: TEHandle; refCon: Longint);
   procedure MyTSMTEPostUpdateProc(textH: TEHandle; fixLen: Longint;
         inputAreaStart, inputAreaEnd: Longint;
         pinStart, pinEnd: Longint; refCon: Longint);
```

C Summary

Constants

```
// signature, interface types

enum  {
   kTSMTESignature     = 'tmTE',
   kTSMTEInterfaceType = kTSMTESignature,
   kTSMTEDialog        = 'tmDI'
};

// Gestalt

enum  {
   gestaltTSMTEAttr    = kTSMTESignature,
   gestaltTSMTEPresent = 0,
   gestaltTSMTEVersion = 'tmTV',
   gestaltTSMTE1       = 0x100
};

// update flag for TSMTERec

enum  {
   kTSMTEAutoScroll    = 1
};
```

Data Types

```
// the following proc ptr declarations come with the usual complements of
// universal proc ptr declarations and related routines

typedef pascal void (*TSMTEPreUpdateProcPtr)(TEHandle textH, long refCon);

typedef pascal void (*TSMTEPostUpdateProcPtr)(TEHandle textH, long fixLen,
      long inputAreaStart, long inputAreaEnd,
      long pinStart, long pinEnd, long refCon);

struct TSMTERec {
   TEHandle            textH;
   TSMTEPreUpdateUPP                   preUpdateProc;
   TSMTEPostUpdateUPP  postUpdateProc;
   long                updateFlag;
   long                refCon;
};

typedef struct TSMTERec TSMTERec, *TSMTERecPtr, **TSMTERecHandle;

struct TSMDialogRecord {
   DialogRecord        fDialog;
   TSMDocumentID       fDocID;
   TSMTERecHandle                      fTSMTERecH;
   long                fTSMTERsvd[3];
};

typedef struct TSMDialogRecord TSMDialogRecord, *TSMDialogPeek;
```

Application-Defined Routines

```
pascal void MyTSMTEPreUpdateProc(TEHandle textH, long refCon);
```

```
pascal void MyTSMTEPostUpdateProc(TEHandle textH, long fixLen,
      long inputAreaStart, long inputAreaEnd,
      long pinStart, long pinEnd, long refCon);
```

Further Reference:

- *Inside Macintosh: Text,* Text Services Manager
- Sample Code: “InlineInputSample”