This chapter describes how text-processing applications can communicate flexibly and efficiently with utilities that provide services to those applications. Applications that need input methods, spell-checking, hyphenation, and so forth can use the Text Services Manager to search for, obtain information about, and communicate with those utilities. Utilities can use the Text Services Manager to request actions and information from applications, and to send data to them.

Read this chapter if you are developing or enhancing an application to use text services. In particular, if you want your application to support text input in a 2-byte script system, you should use the Text Services Manager. Your application will then work with multiple script systems and many input methods.

Read this chapter if you are writing or adapting a utility that provides a text service such as text input. Utilities that work with the Text Services Manager are called *text service components.* If your utility is a text service component, it will be able to communicate with a wide range of applications.

Before reading this chapter, read the chapter "Introduction to Text on the Macintosh" in this book. To use this chapter, you should also be familiar with the Apple Event Manager and the Component Manager. For details on the Apple Event Manager, see *Inside Macintosh: Interapplication Communication*. For more on the Component Manager, see *Inside Macintosh: More Macintosh Toolbox*.

This chapter refers to routines, constants, and data structures from QuickDraw, the Event Manager, the Window Manager, the Menu Manager, and the Process Manager. For details on QuickDraw, see *Inside Macintosh: Imaging*. For more on the Event Manager, Window Manager, and Menu Manager, see *Inside Macintosh: Macintosh Toolbox Essentials*. For information on the Process Manager, see *Inside Macintosh: Processes*.

This chapter first provides a brief introduction to text services in general, input methods in particular, and the Text Services Manager itself. If you are writing an application, it then discusses how you can

- use the Text Services Manager routines for client applications, to send information to text service components
- implement the text-service Apple event handlers in your client application, to receive information from text service components
- communicate directly with the Component Manager and text service components, if your application's special needs require you to bypass the Text Services Manager

If you are writing a text service component, this chapter discusses how you can

- implement the text service component routines, so that the Text Services Manager and client applications can request the text services you provide
- use the Text Services Manager routines for text service components, to send information to client applications and the Text Services Manager

About Text Services

The Text Services Manager is the part of Macintosh system software that maintains communication between applications that need text services and utility programs that provide them. The Text Services Manager exists so that these two types of programs can work together without needing to know anything about each others' internal structures or identities.

A **text service** is a specific text-handling task such as spell-checking, hyphenation, and handling input of complex text. A **text service component** is a utility program that uses the Text Services Manager to provide a text service to an application. Text service components are registered components with the Component Manager, as described in the Component Manager chapter of *Inside Macintosh: More Macintosh Toolbox*.

A **client application** is a text-processing program that uses the Text Services Manager to request a service from a text service component. To accomplish this, a client application needs to make the Text Services Manager aware of its existence and needs to make specific Text Services Manager calls during execution.

In principle, text services can include many different types of tasks. However, only one type of text service is currently defined: text input. This chapter describes how to work with any type of text service component, and how to create any type of text service component, but it emphasizes input methods. It also points out the ways in which input methods are handled differently from other types of text service components.

About Input Methods

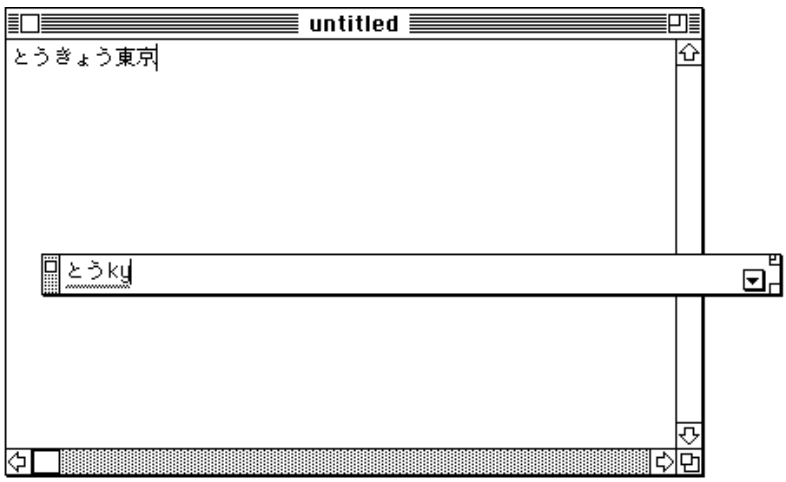
An **input method** is a facility that automatically converts phonetic or syllabic characters into ideographic or other complex representations. It permits use of a standard keyboard to generate the thousands to tens of thousands of different characters needed by some languages. Text input in Japanese, Chinese, and Korean usually requires an input method.

For example, text input in the Japanese script system requires software for transcribing Romaji (phonetic Japanese using Roman characters) or Hiragana (syllabic Japanese) into ideographic Kanji (Chinese characters). Each Kanji character may correspond to more than one possible Hiragana sequence, and vice versa. The input method must grammatically parse sentences or clauses of Hiragana text (which has no word separations) and select the best combination of Kanji and Hiragana characters to represent that text.

Chinese text input is similar to Japanese, in that a conversion from Pinyin (Roman) or Zhuyinfuhao (phonetic) to ideographic Hanzi (Chinese characters) is required. Korean text input requires conversion from Jamo (phonetic) to non-ideographic Hangul (complex clusters of Jamo).

Bottomline input allows the user to type text into a special **floating input window**—usually displayed in the lower portion of the screen—where conversion is to take place. The floating input window typically appears whenever the user starts typing characters. See Figure 7-1.

Figure 7-1 Bottomline input with a floating input window



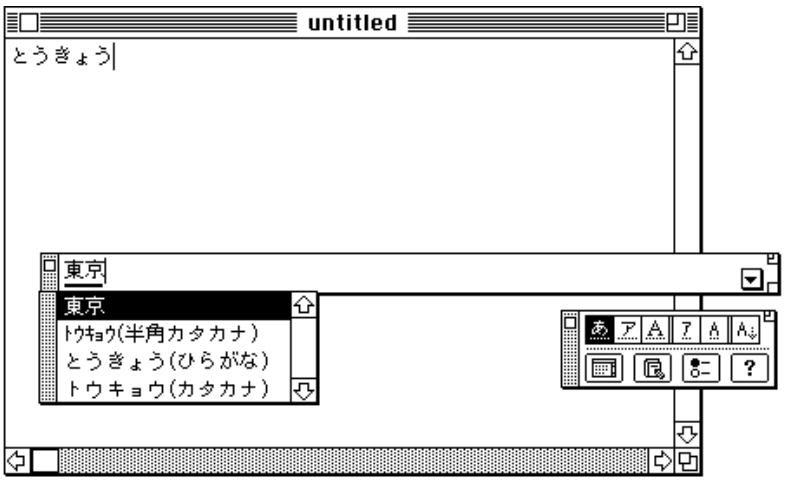
Inline input is an input method in which conversion of characters takes place at the current line position in the application where the text is intended to appear. This allows the user to type text directly into the application window and requires no separate input window. Inline input is the principal example of the kind of text service supported by the Text Services Manager. See Figure 7-2.

Figure 7-2 Inline input

With either bottomline input or inline input, the user can usually type Roman characters or characters of another subscript. Figure 7-3 shows an example of a floating palette, with which the user can select whether text entry is to be in 1-byte or 2-byte Romaji, Katakana, or Hiragana. The user presses a key such as the Space bar to initiate conversion from the input characters to the final characters.

The input method is often extended so that characters may be converted in extremely precise ways. For example, in the Japanese script system, when Hiragana text is converted to Kanji, the user has the option of changing any individual phrase: lengthening it, shortening it, or selecting different possible interpretations. Figure 7-3 shows a scrolling list of additional conversion options displayed next to the converted text in a floating input window. Only after the user is satisfied with the conversion and presses the Return key is the text actually sent to the application.

Figure 7-3 Displaying conversion options for bottomline input



Input methods commonly rely upon one or more dictionaries to perform conversion. The main dictionary lists all standard conversion options for any valid syllabic or phonetic input. Besides using the main dictionary, users can add specialized dictionaries, such as legal or medical dictionaries, to extend the range of the input method. See the chapter "Dictionary Manager" in this book for more information.

About the Text Services Manager

The Text Services Manager links text service components to client applications that use text services. When a client application requests a service from the Text Services Manager, the Text Services Manager routes the request to a text service component associated with that application. The text service component processes the request and may send text or other information back to the Text Services Manager, which passes it on to the client application through an Apple event.

An application that explicitly uses the Text Services Manager is called a **TSM-aware application.** An application that does not make calls to the Text Services Manager is called non-TSM-aware. A non-TSM-aware application can still make indirect use of some services of the Text Services Manager; see "Floating Input Windows" on page 7-13.

The Text Services Environment

The text services environment is a structure for the efficient flow of information between client applications and text service components. It allows client applications to obtain text services without having to know anything about the specific text service components performing them. Likewise, it allows text service components to perform their services without having to know anything about the specific client applications making the requests.

The text services environment consists of a client application, a text service component, the Apple Event Manager, the Component Manager, and the Text Services Manager. For a client application to work within the text services environment, it must

- call the routines of the Text Services Manager application interface described under "Text Services Manager Routines for Client Applications" on page 7-48. By using these application-level routines, a client application becomes TSM-aware and communicates with other parts of the environment.
- implement handlers for the Apple events described under "Apple Event Handlers Supplied by Client Applications" on page 7-65. A client application receives text and other information from a text service component through Apple events.

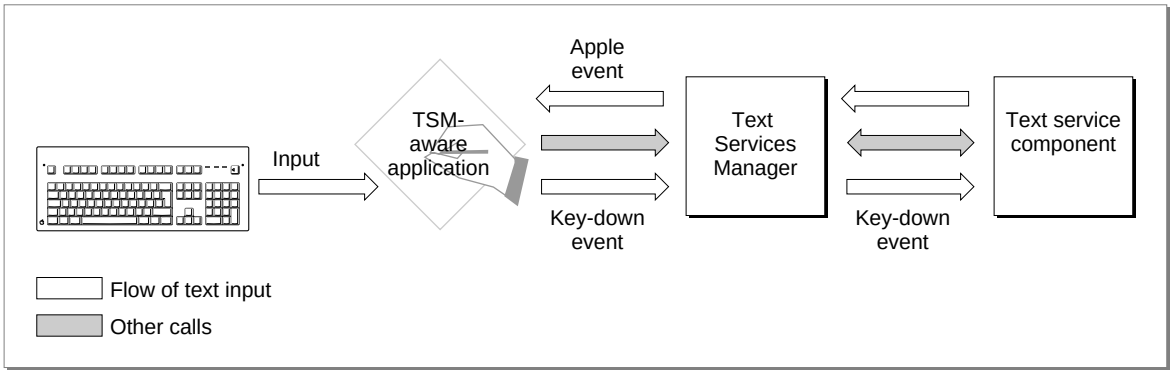
For a text service component to work within the text services environment, it must

- register as a component with the Component Manager
- call the routines of the Text Services Manager component interface described under "Text Services Manager Routines for Components" on page 7-77
- implement the component-level text service component routines described under "Text Service Component Routines" on page 7-84

Figure 7-4 shows some of the flow of information in the text services environment when a TSM-aware application uses a text service component. Application-level calls that an application makes to the Text Services Manager application interface are converted to component-level calls that are passed to an individual text service component. The text service component in turn makes calls to the Text Services Manager component interface; those calls are converted to Apple events that are passed on to the application.

The Text Services Manager controls the overall process by keeping track of which text service components are available to a given application and which application is to receive data from a given text service component. The Text Services Manager communicates with text service components through the Component Manager; applications that have special needs can likewise communicate directly with individual text service components by calling the text service component routines.

Figure 7-4 How a TSM-aware client application uses the Text Services Manager



IMPORTANT

The event-handling structure of the Text Services Manager requires that the low-memory global variable `SEvtEnb` be nonzero. If your application sets `SEvtEnb` to 0 to force the Event Manager function `SystemEvent` to always return a value of `FALSE`, text service components do not function correctly. See *Inside Macintosh: Macintosh Toolbox Essentials* for more information on the `SystemEvent` function and the `SEvtEnb` global variable. ▲

The Text Services Manager and Input Methods

Although the Text Services Manager can work with any type of text service component, it provides several features specific to input methods for 2-byte script systems. The Text Services Manager synchronizes the current input method with the current keyboard script. For example, if the user changes from a Japanese to a Chinese font, the application changes the keyboard script to Chinese and the Text Services Manager then switches the current input method from Japanese to Chinese as well. Unlike with other text services, the Text Services Manager opens and closes input methods, and takes care of their menu handling.

Inline Input

A principal feature of the Text Services Manager is its support for inline input. Figure 7-4 shows how information flows through the Text Services Manager when a TSM-aware application uses it for inline input. The application passes key-down events to the text service component; the text service component sends text and messages back to the application with Apple events. Events, messages, and requests for service between the application and the text service component all pass through the Text Services Manager.

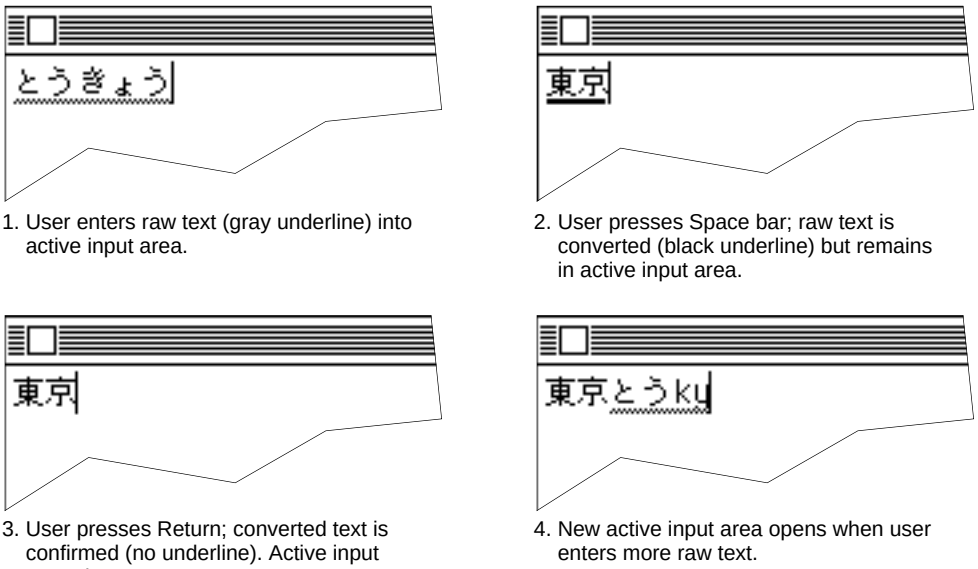
For inline input, the Text Services Manager offers routines that let client applications and text service components communicate about what happens in the **active input area**—the portion of the screen in which the user enters text and where the text service component displays converted text. The client application and the text service component share control over the active input area.

The active input area is almost like a small window with invisible borders inside of the application's document window. It replaces the insertion point in the document, but it can be any width; it can even occupy more than one entire line of text. Text within the active input area can have its own font and size, different from that of body text. Text within the active input area can even scroll out of sight if there is more text than can fit in the space allotted for it in the active input area.

The application is responsible for determining the location and size of the active input area, and for drawing and highlighting all text within it. The text service component is responsible for accepting user input (as key-down events), for converting input text to final text, and for telling the application what characters to draw—and what characters to accept as confirmed—at every step of the way. The text service component can also instruct the application to scroll certain parts of the active input area into view, if necessary.

The text service component processes the user input, called **raw text,** as it is entered. The text service component first has the application draw the text on the screen as entered. Then it **converts** the raw text, translating it from phonetic or syllabic to ideographic or complex syllabic characters. Finally, it **confirms** the converted text upon user approval of the conversion. By convention in some script systems, a text service component converts text when the user presses the Space bar after entering a sequence of characters, and confirms the converted text when the user presses Return to accept the conversion. See Figure 7-5. (In Korean, conversion happens continuously and automatically, and confirmation happens by convention when the user presses either Return or the Space bar.)

Figure 7-5 Entering, converting, and confirming text in an active input area



1. User enters raw text (gray underline) into active input area.
2. User presses Space bar; raw text is converted (black underline) but remains in active input area.
3. User presses Return; converted text is confirmed (no underline). Active input area closes.
4. New active input area opens when user enters more raw text.

The text service component continually removes the confirmed input from the active input area and sends it to the application for storage in its text buffer. The text service component uses Apple events for this purpose, and for notifying the application of every character (raw, converted, or confirmed) that needs to be drawn or highlighted within the active input area.

In a number of situations, a client application may need to initiate the confirmation of input in progress. For example, if a user switches input methods, makes a menu selection, or selects text outside the active input area, the user has implicitly requested confirmation of the existing text. The client application needs to inform the text service component so it can confirm all text, whether raw or converted, in the active input area. The client application can make that request through a call to the Text Services Manager.

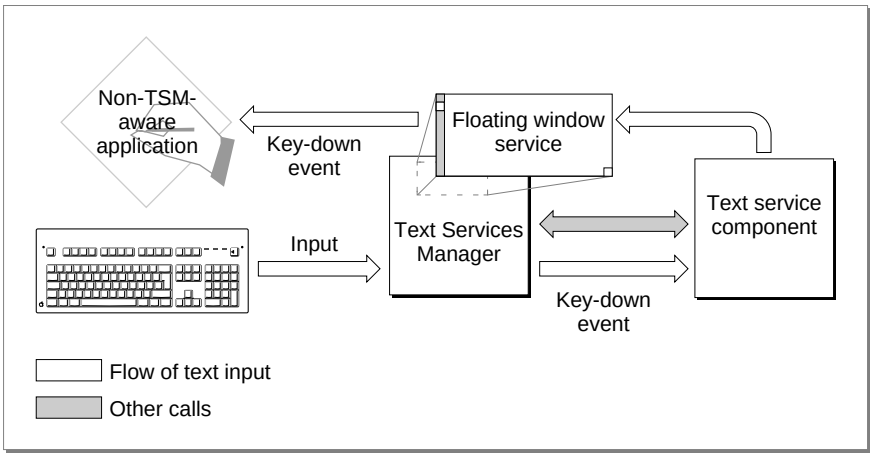
Floating Input Windows

The Text Services Manager also provides a service to facilitate the use of an input window for text entry and conversion when inline input is not supported by the application or not desired by the user. This floating input window is a standard bottomline input window: it usually appears in the lower portion of the screen, although the user can drag it to any location. Once the user's text has been converted correctly in the window, it is sent to the application.

The Text Services Manager's floating input window is mainly for use with applications that are not TSM-aware. See Figure 7-4. The input window uses the **floating window service,** a part of the Text Services Manager. It works this way:

1. The Process Manager intercepts key-down events and passes them to the Text Services Manager.
2. The Text Services Manager passes them to the appropriate input method for processing.
3. The input method then passes the processed text back to the Text Services Manager. The floating window service displays the text in a floating input window.
4. When the user is finished with the text, the floating window service passes the processed text back to the client application through standard key-down events (not Apple events).

Figure 7-6 How a non-TSM-aware application uses the Text Services Manager



In this way the Text Services Manager can provide an input method text service component for applications that have no knowledge of the text services environment.

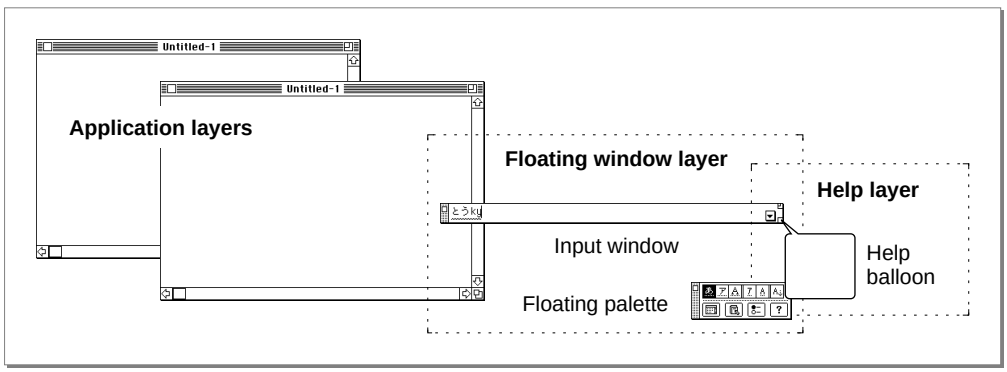
TSM-aware applications should normally use inline input. However, the Text Services Manager does allow TSM-aware applications to use a floating input window. Users may prefer bottomline input if the size of the text displayed in the document makes reading the characters difficult.

Floating Utility Windows

Floating windows are useful for more than just text entry. Input methods can use the Text Services Manager floating window service to create utility windows—floating windows that display palettes or present lists of choices to the user. For example, most Japanese input methods let a user set the input mode to either 2-byte Hiragana, 1-byte Hiragana, 2-byte Romaji, or 1-byte Romaji. In the past, users selected these modes from controls inside the input method's input window. Now, since the system provides a standard floating input window for non-TSM-aware applications as well as for TSM-aware applications that request it, input methods should offer mode selection in a separate floating palette. Figure 7-3 on page 7-8 shows an example of a floating palette window used with bottomline input; Figure 7-9 on page 7-33 shows the same palette used with inline input.

Figure 7-7 illustrates the window-layer organization provided by the Text Services Manager. A floating window, whether an input window or a utility window, is always in front of all application windows but behind any help balloons.

Figure 7-7 Floating window service layer



About Text Service Components

Text service components are components as defined and used by the Component Manager. They have a specific structure, interface, and manner of execution. For more information on components, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*. This section briefly describes the component description record, a data structure associated with a text service component.

The *component description record*, maintained by the Component Manager for each registered component, identifies the characteristics of the component, including the nature of services provided by the component and the manufacturer of the component. It is filled out by the text service component at initialization.

The `ComponentDescription` data type defines the format of the component description record:

```
TYPE ComponentDescription =
   RECORD
      componentType:       OSType;  {command set ID}
      componentSubType:    OSType;  {specifies flavor}
      componentManufacturer:
                           OSType;  {vendor ID}
      componentFlags:      OSType;  {control flags}
      componentFlagsMask:  OSType;  {mask for control flags}
   END;
```

Field descriptions

`componentType` For text service components, this field contains the interface type. The **interface type** specifies the set of Apple events and component commands associated with the text service component. Currently, all text service components have the same interface type, `kTextService`, whose associated 4-character tag is `'tsvc'`. To obtain a list of all available text service components, a client application can specify the value `kTextServices` in the `componentType` field when calling the Component Manager routine `GetServiceList`.

`componentSubType`
For text service components, this field contains the text service component type. The **text service component type** specifies the function and optionally a set of additional routines and data structures associated with that particular kind of text service component. Currently, only one text service component type is defined, `'inpm'`, specifying an inline input method.

`componentManufacturer`
The identification number of the manufacturer of this particular text service component.

`componentFlags` Four bytes that contain component-specific information. See Figure 7-8:

Bits 0–7 contain the language code (as unsigned 8 bits).

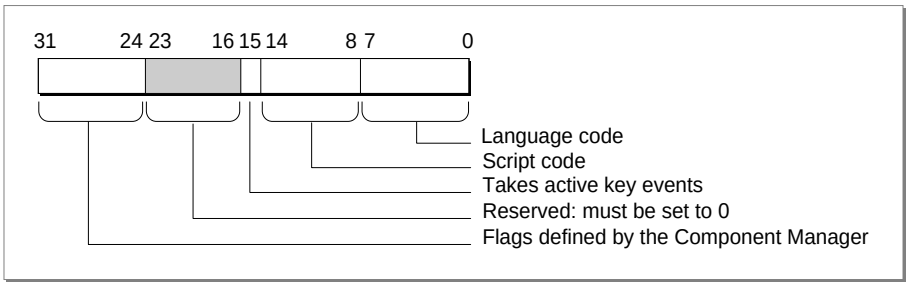
Bits 8–14 contain the script code (as unsigned 7 bits).

Bit 15 indicates whether the text service component takes active events. When bit 15 = 1, the text service component is interactive and accepts user events. When bit 15 = 0, the text service component is not interactive—that is, it only supplies batch services.

Apple has reserved bits 16–23, so text services must set them to 0.

The Component Manager defines bits 24–31.

Figure 7-8 The format of the `componentFlags` field of the component description record



`componentFlagsMask`
Four bytes that contain values used to affect the `componentFlags` field. This field should be 0 in the component description record for any text service component.

For example, an input method for the Japanese script system might assign the following values to the `componentType`, `componentSubType`, and `componentFlags` fields of component description record.

```
cd: ComponentDescription;
cd.componentType     := kTextService;        {'tsvc'}
cd.componentSubType  := kInputMethodService; {'inpm'}
cd.componentFlags    := $0000810B;  {Japanese script & language }
                                    { --takes user events}
```

Using the Text Services Manager (for Client Applications)

This section describes how your client application can use the Text Services Manager application interface to communicate with text service components, how it can use Apple event handlers to receive information from text service components, and how it can communicate directly with text service components—bypassing the Text Services Manager altogether—for special purposes.

Testing for the Availability of the Text Services Manager

Use the `Gestalt` environmental selector `gestaltTSMgrVersion` to determine whether the Text Services Manager is available. The `Gestalt` function returns a 32-bit value indicating which version of the Text Services Manager is installed.

For more information on the `Gestalt` function, see the Gestalt Manager chapter in *Inside Macintosh: Operating System Utilities*.

Calling the Text Services Manager

The application interface to the Text Services Manager consists of application-level calls that your client application uses to send information to text service components by way of the Text Services Manager. They are documented in detail under "Text Services Manager Routines for Client Applications" on page 7-48. The Text Services Manager maps many of those calls to equivalent component-level calls to text service components. Those text service component routines are described under "Text Service Component Routines" on page 7-84.

This section describes how your client application can use the application interface to the Text Services Manager to

- prepare for communication with the Text Services Manager
- create an internal record called a TSM document
- make text services other than text input available to the user
- activate and deactivate a TSM document
- give text service components a chance to handle events, respond to menu selections, and set the shape of the cursor
- explicitly confirm text within the active input area
- terminate communication with the Text Services Manager

Initializing as a TSM-Aware Application

If your client application plans to use any of the Text Services Manager application-interface routines, it must call `InitTSMAwareApplication` at startup, immediately after calling the rest of the Toolbox initialization routines. See Listing 7-1.

Listing 7-1 Initializing as a TSM-aware application

```
FUNCTION Initialize: OSErr;
VAR
   myErr:   OSErr;
BEGIN
   InitGraf(@thePort);
   InitFonts;
   InitWindows;
   InitMenus;
   TEInit;
   InitDialogs(NIL);
   InitCursor;

   IF (InitTSMAwareApplication = noErr) THEN
      Initialize := DoNew;       {application routine that }
                                 { creates window & TSM document}
END;
```

The Text Services Manager records the fact that your client application is TSM-aware, and allocates any private data storage as necessary.

Creating a TSM Document

Your client application needs to create an internal record called a **TSM document** (defined by the `TSMDocument` data type) before it can use any services provided through the Text Services Manager. A TSM document is a private data structure that is associated with each of your application's documents that use a text service. You cannot access the TSM document record directly. You call the `NewTSMDocument` function to instruct the Text Services Manager to create the TSM document. The Text Services Manager returns a TSM document ID, an identifier that you supply in subsequent calls to the Text Services Manager.

Typically, you create a TSM document for each window that your application uses. Use the `supportedInterfaceTypes` array to indicate which text service interfaces you support. Currently only one interface is defined—`'tsvc'`, the component type for text services components. Pass any data you like in the `refcon` parameter to the call. The Text Services Manager returns the `refcon` value in the `keyAETSMDocumentRefcon` parameter of any Apple event sent to your application. You can then use the `refcon` value to determine which TSM document and window the Apple event belongs to.

Listing 7-2 shows the sample application's `DoNew` function, which is called from the initialization routine presented in Listing 7-1. The call to `NewTSMDocument` specifies that the application supports one interface type (`kTextService`). `NewTSMDocument` opens the default input method for the current keyboard script, assigns it to this document, and returns the TSM document ID in the `idocID` parameter. The routine makes use of a modified window record (type `MyWindowRecord`) that is a standard window record with an additional field for holding the TSM document ID.

Listing 7-2 Creating a new TSM document and associating it with a window

```
FUNCTION DoNew: OSErr;
VAR
   wRecordPtr:       myWindowPtr;
   window:           WindowPtr;
   supportedTypes:   InterfaceTypeList;
   myErr:            OSErr;
BEGIN
   supportedTypes[0] := kTextService;
                              {allocate storage for window record}
   wRecordPtr := myWindowPtr(NewPtr(sizeof(myWindowRecord)));
   IF wRecordPtr <> NIL THEN
   BEGIN
      IF gColorQDAvailable THEN
         window := GetNewCWindow(kWINDResID, Ptr(wRecordPtr),
                                 WindowPtr(-1))
      ELSE
         window := GetNewWindow(kWINDResID, Ptr(wRecordPtr),
                                 WindowPtr(-1));
      IF window = NIL THEN                    {couldn't get window}
      BEGIN
         DisposePtr(Ptr(wRecordPtr));                 {clean up}
         DoNew := kWindowFailed;
         Exit(DoNew);
      END;
      myErr := NewTSMDocument(1, supportedTypes,
                              wRecordPtr^.idocID,
                              LongInt(wRecordPtr));
   END;
   {do other window intialization, like creating scroll bars}
   DoNew := myErr;
END;
```

Making Text Services Available to the User

Text services that are input methods are always displayed in the keyboard menu. System software takes care of that; your application does not need to list input methods. However, your application may wish to provide a menu or scrolling list to display other types of available text services. (Note that, currently, no text services other than input methods are available. This capability is provided for future extensibility.)

To obtain a list of the available text services on the user's system, call the `GetServiceList` function. You pass it an array of interface types (to indicate the types of services you want returned in the list) and a pointer to a data structure (to hold the list). The function returns the number of available components, and a name and component identifier for each one.

Because text service components can be registered or unregistered at any time, your client application should periodically call either `GetServiceList` or the Component Manager function `GetComponentListModSeed` to see if the list of registered text service components may have changed.

IMPORTANT
If your client application displays a list or menu of text service components, do not show input methods. They are already displayed in the Keyboard menu. To show them in two places would be confusing to users. ▲

The Text Services Manager automatically opens input methods; your client application does not have to open them. You do have to explicitly open all other types of text services, however. If the user chooses a text service from a menu or list that you have displayed, you need to open that text service.

You call the `OpenTextService` function to associate the text service component with the current TSM document. `OpenTextService` then returns a valid component instance to indicate that the text service component has been opened and initialized.

Whenever a user wishes to close a text service component that you have opened, call the `CloseTextService` function.

Activating and Deactivating a TSM Document

To notify the Text Services Manager that a window in your client application associated with a TSM document has been activated, and that you are ready to use a text service component, use the `ActivateTSMDocument` function.

Listing 7-3 shows how to handle activating and deactivating a TSM document. You specify the document using the ID assigned to it when it was created (with the `NewTSMDocument` function). This routine, like the previous samples, assumes that the application has an extended window record with a field, `idocID`, that contains the TSM document ID.

Listing 7-3 Activating and deactivating a TSM document

```
PROCEDURE DoActivate(window: WindowPtr; becomingActive: Boolean);
VAR
   myErr:   OSErr;
BEGIN
   IF becomingActive THEN
      myErr := ActivateTSMDocument(MyWindowPtr(window)^.idocID)
   ELSE
      myErr := DeactivateTSMDocument(MyWindowPtr(window)^.idocID);
END;
```

When the Text Services Manager receives an `ActivateTSMDocument` call, it deactivates the currently active TSM document (if it hasn't already been explicitly deactivated) and stores the new document as the currently active TSM document. If the specified text service component for the document has a menu, the Text Services Manager inserts the menu into the menu bar as an application or system menu.

When a window in your client application associated with a TSM document has been deactivated, you should call the `DeactivateTSMDocument` function. The Text Services Manager in turn calls the text service component function `DeactivateTextService` for any text service components associated with the TSM document being deactivated.

Input-method text services are handled in a special way: the identity of the input method of the deactivated document is retained by the Text Services Manager, and compared with the input method used by the next *activated* document. If the newly active document uses the same input method, the Text Services Manager will simply activate the new instance of the same input method. If the documents use different input methods, the previous input method is then closed, and any windows belonging to it are closed and menus are removed. The new input method is then activated. Not closing an input method until it is actually unneeded avoids extra removal and immediate redisplay of input method palettes and menus.

Passing Events, Menu Selections, and Cursor Setting

Whenever your client application receives an event from the Event Manager function `WaitNextEvent`, you need to give each text service component an opportunity to handle that event, if appropriate. Use the `TSMEvent` function to let the Text Services Manager dispatch the events to the correct text service component. You provide a pointer to the event record containing the event. The Text Services Manager passes the event in turn to each component associated with the currently active TSM document, starting with input methods. If the event is handled by a component, `TSMEvent` returns `TRUE` and the event is changed to a null event. If the event is not handled, `TSMEvent` returns `FALSE` and you are responsible for handling the event.

Listing 7-4 is a partial example of an event handler in which the application passes events to the Text Services Manager for routing to text service components. If no text service component handles an event, the application handles it. The global variable

`gUsingTSM` is `TRUE` if the Text Services Manager is present and the application is making use of it.

Listing 7-4 Passing events to a text service component

```
PROCEDURE MyDoEvent(event: eventRecord);
VAR
   handledByTS:   Boolean;
   gotEvent:      Boolean;
BEGIN
   WHILE TRUE DO
   BEGIN
      IF gHasWaitNextEvent THEN
      BEGIN
         gotEvent := WaitNextEvent(everyEvent, event,
                                   kSleep, NIL);
         handledByTS := FALSE;
         IF (gUsingTSM AND gotEvent) THEN
            handledByTS := TSMEvent(event);
      END;
      IF gotEvent AND (NOT handledByTS) THEN

         {process event in normal way}
         ;
   END;
END;
```

Whenever a user chooses a menu item, it may be from a text service component's menu; your application must therefore give the text service component a chance to respond. (This situation occurs only with text service components that are not input methods.) To do this, use the `TSMMenuSelect` function with the result from the Menu Manager function `MenuSelect` in the `menuResult` parameter. If `TSMMenuSelect` returns `TRUE`, then the text service component has handled the menu selection, so your client application does not need to do so. However, your application is still responsible for removing the highlighting from the menu title after the selection has been handled.

Your client application is generally responsible for setting the cursor to an appropriate shape. However, the text service component may have its own cursor requirements when the cursor is within the boundaries of its windows or palettes. To allow a text service component to set the cursor, use the `SetTSMCursor` function. Call it whenever you would normally set the cursor yourself. If `SetTSMCursor` returns `TRUE`, the cursor is either on a text service component window or on the active input area and a text service component has set the cursor. In this case, you should not set the cursor.

Confirming Active Text Within a TSM Document

Normally, an input method text service component ejects finished input from the active input area continually as it processes user events, sending the confirmed text to your application with the Update Active Input Area Apple event.

Circumstances may arise in which you need to confirm input in progress before the text service component ejects it (that is, before the user presses Return). If, for example, the user clicks the mouse in text outside the active input area, that constitutes an implicit user acceptance of the text in the active input area. You should explicitly terminate any active input and save the text that is in the active input area by calling the `FixTSMDocument` function. The text service component sends the confirmed text to your application and empties the active input area.

Listing 7-5 shows what happens when the user clicks the go-away box of the active document window after entering some text in the active input area. The global variable `gIDocID` represents the ID of the active TSM document.

Listing 7-5 Confirming text in an active input area

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:       Integer;
   theWindow:  WindowPtr;
   myErr:      OSErr;
BEGIN
   part := FindWindow(event.where, theWindow);
   CASE part OF
      inContent:
         DoContentClick(theWindow, event);

      inDrag:
         DragWindow(theWindow, event.where,
                                theWindow^.portRect);

      inGoAway:
         IF TrackGoAway(theWindow, event.where) THEN
         BEGIN
            myErr := FixTSMDocument(gIDocID);   {confirm text}
            DoActivate(theWindow, FALSE);    {deactivate window}
            HideWindow(theWindow);              {put it away}
            gVisible := FALSE;
         END;
   END;
END;
```

Deleting a TSM Document

When your client application closes a document window and no longer needs its associated TSM document, it needs to call the `DeleteTSMDocument` function to inform the Text Services Manager that the TSM document should be deleted.

The Text Services Manager closes the text service components for the specified TSM document by calling the Component Manager `CloseComponent` function for each open text service component. It then disposes of the internal TSM document record for the specified TSM document.

Closing Down as a TSM-Aware Application

To let the Text Services Manager perform needed housekeeping chores when your application has closed, your client application needs to call `CloseTSMAwareApplication` just before quitting, as shown in Listing 7-6.

Listing 7-6 Closing a TSM-aware application

```
FUNCTION DoQuitApplication: OSErr;
VAR
   myErr:   OSErr;
BEGIN

   {app-specific clean up}

   myErr := CloseTSMAwareApplication;  {ignore the error}
   ExitToShell;
END;
```

Requesting a Floating Input Window for Text Entry

Your client application may need to provide for users who prefer to enter text using a floating input window instead of entering text directly in the line of a document. For example, when the text font size is too small for reading ideographic characters, too big for convenient entry directly into the document window, or is being greeked, users may prefer a floating input window.

Your client application calls the `UseInputWindow` function with the `useInputWindow` parameter set to `TRUE` to display a floating input window for the TSM document you specify in the `idocID` parameter to the call. To display floating input windows for all documents associated with your application, you set the `idocID` parameter to `NIL` and the `useInputWindow` parameter to `TRUE`. To return to inline input, call `UseInputWindow` with the `useInputWindow` parameter set to `FALSE`.

Associating Input Methods With Scripts and Languages

If you use the application-interface routines, the Text Services Manager automatically associates a default input method with your TSM document every time the current script and language change. Although it is unlikely that it would ever need to, your client application can use Text Services Manager routines to control that automatic association.

The Operating System uses the `GetDefaultInputMethod` and `SetDefaultInputMethod` functions to associate an input method with a given script and language. When the user uses the Keyboard menu, Keyboard control panel, or other device for controlling input method preferences, these functions establish permanent associations (they last across restarts).

The Text Services Manager maintains a current text service language that it uses to synchronize input methods to the current script system and language. The Operating System calls the `SetTextServiceLanguage` function when the user switches the keyboard script, and the floating window service calls the `GetTextServiceLanguage` function to determine the text service language.

These routines make use of the script-language record, described under "Identifying the Supported Scripts and Languages" on page 7-42.

If your client application uses the Text Services Manager application-interface routines, the Text Services Manager automatically synchronizes the input method to the current text service language and there is no need to make the calls described here. If your client application bypasses the Text Services Manager and uses the text service component routines, the Text Services Manager does not provide automatic input method synchronization and you may have to make some of these calls yourself. See "Direct Access to Text Service Components" on page 7-36 for more information on the Component Manager and on how to communicate directly with text service components.

Handling Text Service Apple Events

Text service components send information to your client application through Apple events. To communicate with an input method text service component, you need to implement Apple event handlers that

- receive raw, converted, or confirmed text from the input method, update the active input area, and highlight text appropriately
- convert screen location (in global coordinates) to text offset (in the active input area or in the application's text buffer), so that the input method can, for example, adjust the caret position or cursor display to reflect the text beneath the cursor
- convert text offset to screen location, so that the input method can, for example, place a list of conversion options next to a particular section of raw text
- respond to the input method's request to show or hide a floating input window

Each Apple event contains two required parameters:

- The `keyAETSMDocumentRefcon` parameter is filled in by the Text Services Manager. It tells the application which TSM document is affected by the Apple event.
- The `keyAEServerInstance` parameter is filled in by the text service component, and identifies the component that is sending the Apple event.

Other parameters are specific to each Apple event, and are described under "Apple Event Handlers Supplied by Client Applications" on page 7-65.

For general rules for writing Apple event handlers, see the discussion of the Apple Event Manager in *Inside Macintosh: Interapplication Communication*.

Receiving Text and Updating the Active Input Area

The text service component uses the Update Active Input Area Apple event to request that your client application create and update an active input area, including drawing text in the active input area, and accepting confirmed text. For details on active input areas, see "Inline Input" on page 7-11. This Apple event also asks the client application to update a range of text in the active input area and highlight appropriately.

Because your application is responsible for all drawing in the active input area, it receives an Update Active Input Area Apple event whenever the user enters raw text (for example, Romaji for Japanese input), whenever that raw text is converted to an intermediate form (for example, Hiragana), whenever the text is converted (for example, to Kanji), and whenever the converted text is confirmed. The input method also uses this Apple event to instruct your application in how to highlight the various types of text (raw, converted, and so on) within the active input area.

The input method uses the Update Active Input Area Apple event to send additional information to your application, such as current caret position, a range of text that should be scrolled into view if it is not visible, and boundaries of clauses (language-specific groupings of text) that may exist in the active input area.

Listing 7-7 shows a sample handler for the Update Active Input Area Apple event. The handler first receives the input parameters, including the text and the ranges of text to highlight and update. The handler then puts any confirmed text into the application's text buffer.

Listing 7-7 A sample handler for the Update Active Input Area Apple event

```
FUNCTION MyHandleUpdateActive(theAppleEvent: AppleEvent;
                              reply: AppleEvent;
                              handlerRefCon: LongInt): OSErr;
VAR
   theHiliteDesc: AEDesc;
   theUpdateDesc: AEDesc;
   theTextDesc:   AEDesc;
   myErr:         OSErr;
```

```
   returnedType:  DescType;
   script:        ScriptLanguageRecord;
   fixLength:     LongInt;
   refcon:        LongInt;
   textSize:      LongInt;
   actualSize:    LongInt;
   thePinRange:   TextRange;

BEGIN
   {Get the required parameter keyAETSMDocumentRefcon}
   myErr := AEGetParamPtr(theAppleEvent, keyAETSMDocumentRefcon,
                          typeLongInteger, returnedType, @refcon,
                          sizeof(refcon), actualSize);
   IF myErr = noErr THEN
   BEGIN
      {Get the required parameter keyAETheData}
      theTextDesc.dataHandle := NIL;
      myErr := AEGetParamDesc(theAppleEvent, keyAETheData,
                              typeChar, theTextDesc);
   END;
   IF myErr <> noErr THEN
   BEGIN
      MyHandleUpdateActive := myErr;
      Exit(MyHandleUpdateActive);
   END;


   {Get the required parameter keyAEScriptTag}
   myErr := AEGetParamPtr(theAppleEvent, keyAEScriptTag,
                          typeIntlWritingCode, returnedType,
                          @script, sizeof(script), actualSize);
   IF myErr = noErr THEN
      {Get the required parameter keyAEFixLength}
      myErr := AEGetParamPtr(theAppleEvent, keyAEFixLength,
                             typeLongInteger, returnedType,
                             @fixLength, sizeof(fixlength),
                             actualSize);
   IF myErr = noErr THEN
   BEGIN
      {Get the optional parameter keyAEHiliteRange}
      theHiliteDesc.dataHandle := NIL;
      myErr := AEGetParamDesc(theAppleEvent, keyAEHiliteRange,
                              typeTextRangeArray, theHiliteDesc);
   END;
```

```
IF myErr <> noErr THEN
BEGIN
   MyHandleUpdateActive := myErr;
   myErr := AEDisposeDesc(theTextDesc);   {ignore the error}
   Exit(MyHandleUpdateActive);
END;

{Get the optional parameter keyAEUpdateRange}
theUpdateDesc.dataHandle := NIL;
myErr := AEGetParamDesc(theAppleEvent, keyAEUpdateRange,
                        typeTextRangeArray, theUpdateDesc);
IF myErr <> noErr THEN
BEGIN
   MyHandleUpdateActive := myErr;
   myErr := AEDisposeDesc(theTextDesc);   {ignore the error}
   myErr := AEDisposeDesc(theHiliteDesc);
   Exit(MyHandleUpdateActive);
END;

{Get the optional parameter keyAEPinRange}
myErr := AEGetParamPtr(theAppleEvent, keyAEPinRange,
                        typeTextRange, returnedType,
                        @thePinRange, sizeof(thePinRange),
                        actualSize);

MyHandleUpdateActive := myErr;
IF myErr = noErr THEN
BEGIN
   textSize := GetHandleSize(theTextDesc.dataHandle);
   MyHandleUpdateActive := MemError;
   IF MemError = noErr THEN
   BEGIN
      {if the value of keyAEFixLength is -1, the text }
      { contained in the keyAETheData parameter should }
      { completely replace the active input area in }
      { the application window}

      IF fixLength = -1 THEN fixLength := textSize;

      {the application procedure SetNewText handles }
      { updating and confirming the text in the active }
      { input area, highlighting, and scrolling the }
      { specified offsets into view}
```

```
            SetNewText(refcon, script, theTextDesc.dataHandle,
                  textSize, fixLength,
                  TextRangeArrayHandle(theUpdateDesc.dataHandle),
                  TextRangeArrayHandle(theHiliteDesc.dataHandle) );
         END;
      END;
      myErr := AEDisposeDesc(theTextDesc);      {ignore the errors}
      myErr := AEDisposeDesc(theHiliteDesc);
      myErr := AEDisposeDesc(theUpdateDesc);
END;
```

Converting Screen Position to Text Offset

An input method text service component uses the Position To Offset Apple event when it needs to know the byte offset in a text buffer (usually the buffer corresponding to the active input area) corresponding to a given screen position. An input method typically sends the Position To Offset Apple event to your application in response to a mouse-down event. If the event location is in the application window (including the active input area), the input method may want to know which character the event corresponds to, in order to locate the caret or define highlighting.

An input method may also send Position To Offset in response to `SetTSMCursor`, so that it can modify the appearance of the cursor depending on the type of text the cursor passes over.

Your application's handler returns a byte offset and a value indicating whether the screen position is within the active input area. If it is, the offset is measured from the start of the active input area (the leading edge of the first character on the first line). If it is not, the offset is measured from the beginning of the application's body text. The definition of *body text* and the significance of measurements within it are specific to your application; here it means any application text outside of the active input area.

To help the input method more specifically define individual characters, your application can optionally return an indication as to whether the position corresponds to the leading edge or the trailing edge of the glyph corresponding to the character at the indicated offset.

The Position To Offset Apple event is similar in function to the QuickDraw `PixelToChar` function, and returns similar results. Your handler may use `PixelToChar` to get the information it returns to the text service component, or it may use a TextEdit call, as shown in the following code sample.

Listing 7-8 shows a sample handler for the Position To Offset Apple event. The handler first receives the input parameters, then uses the TextEdit function `TEGetOffset` to convert a screen location to text offset. The `TEGetOffset` function is described in the chapter "TextEdit" in this book.

Listing 7-8 A sample handler for the Position To Offset Apple event

```
FUNCTION MyHandlePos2Offset(theAppleEvent: AppleEvent;
                            reply: AppleEvent;
                            handlerRefCon: LongInt): OSErr;
VAR
   myErr:         OSErr;
   returnedType:  DescType;
   refcon:        LongInt;
   currentPoint:  Point;
   clickWindow:   WindowPtr;
   where, part:   Integer;
   oldPort:       GrafPtr;
   offset:        LongInt;
   te:            TEHandle;
   actualSize:    LongInt;
   bodyRect:      Rect;
   dragging:      Boolean;
   isMatch:       Boolean;

BEGIN
   {Get the required parameter TSMDocumentRefcon}
   myErr := AEGetParamPtr (theAppleEvent, keyAETSMDocumentRefcon,
                           typeLongInteger, returnedType, @refcon,
                           sizeof(refcon), actualSize);
   IF myErr = noErr THEN
      {Get the required parameter keyAECurrentPoint}
      myErr := AEGetParamPtr (theAppleEvent, keyAECurrentPoint,
                              typeQDPoint, returnedType,
                              @currentPoint,
                              sizeof(currentPoint), actualSize);
   IF myErr <> noErr THEN
   BEGIN
      MyHandlePos2Offset := myErr;
      Exit(MyHandlePos2Offset);
   END;

   where := kTSMOutsideOfBody;
   part  := FindWindow(currentPoint, clickWindow);

   {the application function IsWindowForTheAE returns TRUE}
   {if the refcon is associated with the window}

   isMatch := IsWindowForTheAE(refcon, clickWindow);
```

```
IF ((clickWindow = FrontWindow) AND
      isMatch AND (part = inContent)) THEN
BEGIN
   GetPort(oldPort);
   SetPort(clickWindow);

   {convert currentPoint into the local }
   { coordinates of the current grafport}

   GlobalToLocal(currentPoint);

   {the application function GetTheBodyRect returns the}
   {body rect of the window}

   bodyRect := GetTheBodyRect(clickWindow);
   IF PtInRect(currentPoint, bodyRect) THEN
   BEGIN
      where  := kTSMInsideOfBody;


   {the application function FindTheTEHandle returns the }
   { window's TEHandle. Then the TextEdit function }
   { TEGetOffset returns the offset corresponding to the point}

      te := FindTheTEHandle (clickWindow);
      offset := TEGetOffset(currentPoint, te);

      {The application function IsInsideInputArea returns }
      { TRUE if offset is within the active input area. }
      { It is application's responsibility to remember }
      { the range of the input area.}

      IF IsInsideInputArea(offset, clickWindow) THEN
         where  := kTSMInsideOfActiveInputArea;
   END;

   {get the optional parameter: keyAEDragging}
   dragging := FALSE;
   myErr := AEGetParamPtr (theAppleEvent, keyAEDragging,
                           typeBoolean, returnedType,
                           @dragging, sizeof(dragging),
                           actualSize);
END;
IF myErr <> noErr THEN
```

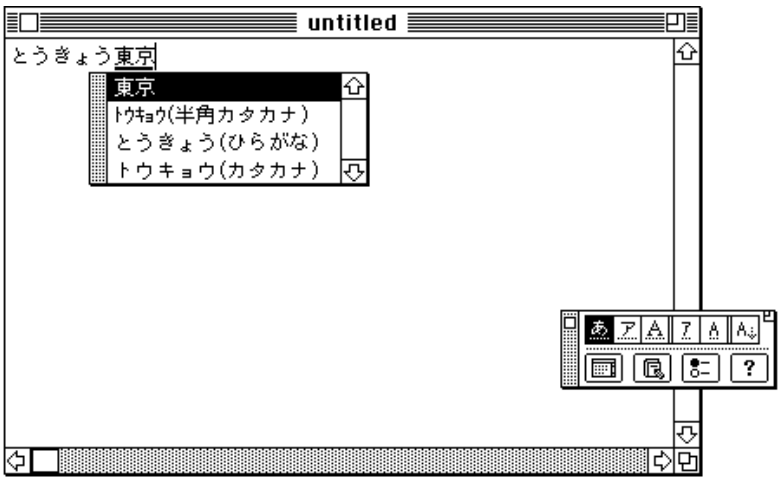
```
   BEGIN
      MyHandlePos2Offset := myErr;
      Exit(MyHandlePos2Offset);
   END;

   {if the parameter keyAEdragging is TRUE and the mouse}
   { position is outside the body text, the application }
   { can scroll the text within the active input area, }
   { rather than returning kTSMOutsideOfBody. The application }
   { procedure HandleScroll is handling the scrolling.}

   IF (dragging = TRUE) AND (where = kTSMOutsideOfBody) THEN
   BEGIN
      HandleScroll(te, offset);
      where := kTSMInsideOfActiveInputArea;
   END;
   SetPort(oldPort);

   {Construct the return parameter keyAEOffset}
   myErr := AEPutParamPtr(reply, keyAEOffset, typeLongInteger,
                          @offset, sizeof(offset));
   IF myErr = noErr THEN
      {Construct the return parameter keyAERegionClass}
      MyHandlePos2Offset := AEPutParamPtr(reply, keyAERegionClass,
                                    typeShortInteger,
                                    @where, sizeof(where))
   ELSE MyHandlePos2Offset := myErr;
END;
```

Converting Text Offset to Screen Position

An input method text service component uses the Offset To Position Apple event when it needs to know the screen position corresponding to a given byte offset in the text buffer for the active input area. An input method typically sends the Offset To Position Apple event to your application when it needs to draw something in a specific spatial relationship with a given character in the active input area. For example, it may need to draw a floating window containing suggested conversion options beside a particular range of raw or converted text. See Figure 7-9.

The text service component supplies a byte offset, measured from the character at the start of the active input area. The application returns a point designating the global coordinates of the caret position corresponding to that offset. Your application may optionally return information about the font, size, and other measurements of the text in the active input area, so that the text service component can more precisely locate the elements it is to draw.

Figure 7-9 Drawing a window with conversion options next to the active input area



The Offset To Position Apple event is similar in function to the QuickDraw `CharToPixel` function, and it returns similar results. Your handler may use `CharToPixel` to get the information it returns to the text service component, or it may use a TextEdit call, as shown in the following code sample.

Listing 7-9 shows a sample handler for the Offset To Position Apple event. The handler first receives the input parameters, then uses the TextEdit function `TEGetPoint` to convert a text offset to a screen location. The `TEGetPoint` function is described in the chapter "TextEdit" in this book.

Listing 7-9 A sample handler for the Offset To Position Apple event

```
FUNCTION MyHandleOffset2Pos(theAppleEvent: AppleEvent;
                            reply: AppleEvent;
                            handlerRefCon: LongInt): OSErr;
VAR
   myErr:         OSErr;
   rtErr:         OSErr;
   returnedType:  DescType;
   offSet:        LongInt;
   refcon:        LongInt;
   actualSize:    LongInt;
   theWindow:     WindowPtr;
   te:            TEHandle;
   oldPort:       GrafPtr;
   thePoint:      Point;
   theFixed:      Fixed;
BEGIN
```

```
{Get the required parameter keyAEOffset}
myErr := AEGetParamPtr(theAppleEvent, keyAEOffset,
                       typeLongInteger, returnedType, @offSet,
                       sizeof(offSet), actualSize);
IF myErr = noErr THEN
   {Get the required parameter TSMDocumentRefcon}
   myErr := AEGetParamPtr(theAppleEvent,
                          keyAETSMDocumentRefcon,
                          typeLongInteger, returnedType,
                          @refcon, sizeof(refcon),
                          actualSize);
IF myErr <> noErr THEN
BEGIN
   MyHandleOffset2Pos := myErr;
   Exit(MyHandleOffset2Pos);
END;
{the application function GetWindowFromRefcon returns the }
{ window which is associated with the refcon}

rtErr := noErr;                              {initialize rtErr}
theWindow := GetWindowFromRefcon(refcon);
IF theWindow = NIL THEN
   rtErr := errOffsetInvalid
ELSE
BEGIN
   {the application function FindTheTEHandle returns the }
   { TEHandle for the window}

   te := FindTheTEHandle(theWindow);

   {the TextEdit function TEGetPoint returns the point }
   { corresponding to the given offset}

   thePoint := TEGetPoint(offSet, te);
   IF (offSet > te^^.teLength) OR (offSet < 0) THEN
      rtErr := errOffsetInvalid
   ELSE IF(PtInRect(thePoint,theWindow^.portRect) = FALSE) THEN
      rtErr := errOffsetIsOutsideOfView
   ELSE
   BEGIN
      GetPort(oldPort);
      SetPort(theWindow);
```

```
      {Convert thePoint into global coordinates}

      LocalToGlobal(thePoint);
      SetPort(oldPort);
   END;

   {construct the return parameter keyAEPoint}
   myErr := AEPutParamPtr(reply, keyAEPoint, typeQDPoint,
                          @thePoint, sizeof(thePoint));
   IF myErr = noErr THEN
      {construct the optional return parameter keyAETextFont}
      myErr := AEPutParamPtr(reply, keyAETextFont,
                          typeLongInteger, @te^^.txFont,
                          sizeof(longInt));
   IF myErr = noErr THEN
   BEGIN
      {construct optional return parameter keyAETextPointSize}
      theFixed := BSL(Fixed(te^^.txSize), 16);
      myErr := AEPutParamPtr(reply, keyAETextPointSize,
                          typeFixed, @theFixed,
                          sizeof(theFixed));
   END;
   IF myErr = noErr THEN
      {construct optional return parameter keyAETextLineHeight}
      myErr := AEPutParamPtr(reply, keyAETextLineHeight,
                          typeShortInteger, @te^^.lineHeight,
                          sizeof(Integer));
   IF myErr = noErr THEN
      {construct optional return parameter keyAETextLineAscent}
      myErr := AEPutParamPtr(reply, keyAETextLineAscent,
                          typeShortInteger, @te^^.fontAscent,
                          sizeof(Integer));
   IF myErr = noErr THEN
   BEGIN
      {construct the optional return parameter keyAEAngle-- }
      {90 = horizontal direction, 180 = vertical direction}
      theFixed := Fixed(90);
      myErr := AEPutParamPtr(reply, keyAETextPointSize,
                          typeFixed, @theFixed,
                          sizeof(Fixed));
   END;
   IF myErr <> noErr THEN
   BEGIN
```

```
            MyHandleOffset2Pos := myErr;
            Exit(MyHandleOffset2Pos);
         END;
      END;
      {Construct the return parameter keyErrorNumber}
      MyHandleOffset2Pos := AEPutParamPtr(reply, keyErrorNumber,
                                          typeShortInteger, @rtErr,
                                          sizeof(rtErr));
END;
```

Showing or Hiding the Input Window

Input methods that work with a floating input window often offer options to the user for either (1) continually displaying the input window, (2) displaying it only as text is being typed in and hiding it immediately after the user confirms it, or (3) leaving the window up for a specified amount of time after confirmation. The Show/Hide Input Window Apple event requests that your client application make the bottomline floating input window either visible or not visible. An input method text service component sends this Apple event whenever it needs to know or change the current state of the window.

This Apple event is for use only by applications that display their own input windows. If your application does not itself control the display of a floating input window, you can ignore this Apple event. If your application uses the Text Services Manager floating window service for bottomline input (by calling `UseInputWindow`), you do not receive this Apple event because it is handled by the Text Sevices Manager.

Direct Access to Text Service Components

Your client application can bypass the Text Services Manager and communicate with text service components directly. Many of the text service component routines correspond in function to the Text Services Manager application-interface routines. It is therefore possible for a client application to use the text service component routines if it needs to exert finer control over its interaction with text service components or if it requires specific kinds of text services or server-specific knowledge. It is not recommended in most cases, because the Text Services Manager is not available to help with dispatching and housekeeping chores.

Calling the Component Manager

If your client application does not use the Text Services Manager, it has to communicate with the Component Manager directly to identify and initialize individual text service components. You can use Component Manager calls to find components, set a default component, get information about components, and open components. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information.

Calling Text Service Components

If your client application calls text service components directly, it uses the text service component routines, a component-level interface described under "Text Service Component Routines" on page 7-84.

After opening a text service component with the `OpenComponent` or `OpenDefaultComponent` function, your client application calls `InitiateTextService` function to instruct the text service component to commence its operations.

To inform a text service component that its associated document window is becoming active or inactive, call the `ActivateTextService` or `DeactivateTextService` function.

You are responsible for adding the text service component menu to your application's menu bar. Furthermore, you are responsible for either disabling the menu or removing it from your menu bar when the text service component becomes inactive. Call the `GetTextServiceMenu` function to obtain menus from each open text service component.

To pass events to text service components, call the `TextServiceEvent` function. You are also responsible for allowing the text service components to control the cursor. Use the `SetTextServiceCursor` function to give the text service component a chance to set the cursor.

When a user makes a selection from the menu for a text service component, call the `TextServiceMenuSelect` function.You should call `TextServiceMenuSelect` right after the Menu Manager routines `MenuSelect` or `MenuKey`.

Before closing the component, call the `TerminateTextService` function to tell the text service component to finish its operations. You should remove the text service component's menu from the menu bar when the text service component is deactivated.

Using the Text Services Manager (for Text Service Components)

This chapter does not describe how to write a text service component. It describes only the interface between text service components and the Text Services Manager. Each text service component has several functions; it must be able to

- perform the tasks for which it was created
- communicate with the Component Manager
- receive calls from the Text Services Manager (or client applications), through the Component Manager
- send calls to the Text Services Manager

How components perform their specific text-handling tasks is beyond the scope of *Inside Macintosh.* How components communicate with the Component Manager is described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

How text service components communicate with the Text Services Manager is described in this section.

The text service component routines are the component-level calls that the Text Services Manager makes to text service components through the Component Manager. See "Text Service Component Routines" on page 7-84 for detailed descriptions of the calls. If you are writing a text service component, it must implement the text service component routines.

Text service components also make calls to the Text Services Manager, to send Apple events to client applications and to request the use of a floating window when needed. See "Text Services Manager Routines for Components" on page 7-77 for detailed descriptions of those component-interface calls.

For a brief discussion of some of the data types associated with text service components, see "About Text Service Components" beginning on page 7-14.

Providing Menus and Icons

If you are writing a text service component, you can have it display its own menu, provide an icon for the title of that menu, and provide icons for the Keyboard menu.

Providing a Text Service Component Menu

Although most user selections and configurations are best made with floating palettes, a text service component may put one menu into the menu bar. For input-method text service components, the menu cannot be hierarchical. Input-method menus appear on the right (system) side of the menu bar, between the Help menu and the Keyboard menu. Menus for non-input-method text service components appear on the left (application) side of the menu bar. See Figure 7-10 on page 7-40.

To create the menu, follow the standard procedures as described in the Menu Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials.* The Text Services Manager installs the menu in the menu bar whenever your component is opened or activated. The application (through the Text Services Manager) passes the menu commands to you for handling when appropriate.

All instances of an input-method text service component must share one menu handle. Therefore, make sure to allocate the handle in the System heap. You can store the menu handle in your component's `refcon` field. See the discussion of the Component Manager `SetComponentRefcon` routine in *Inside Macintosh: More Macintosh Toolbox*.

IMPORTANT

An input-method text service component should never dispose of its menu handle in response to a `TerminateTextService` call (see page 7-86). Any other kind of text service component should always dispose of its menu handle in response to a `TerminateTextService` call. ▲

Using an icon for the menu title

If you wish to have an icon instead of text as the title of your text service component menu, first create a small-icon suite (such as `'kcs#'`, `'kcs4'`, and `'kcs8'`) to represent your menu title. Then, in your menu resource, make the menu title a 5-byte Pascal string (6 bytes total size), with this format:

Byte	Value
0	$05 (length byte for menu string)
1	$01 (invalid character code)
2–5	Handle to icon suite

When the menu is created, the menu bar definition procedure knows from the values of the first 2 bytes that the final 4 bytes are a handle to an icon suite, and the procedure will put the icon in the menu bar. For more on creating icon suites and drawing icons, see the Finder Interface chapter of *Inside Macintosh: Macintosh Toolbox Essentials*. See also *Macintosh Human Interface Guidelines* for design suggestions for color icon families. ◆

Remember these limitations when considering an input-method menu: an input method can put up only one menu, the menu cannot be hierarchical, and the menu can be removed from the menu bar if there is insufficient room for it (on a small screen). It may be more appropriate to use palettes.

Providing Input Method Icons for the Keyboard Menu

Any text service component that provides an input method must supply the following keyboard icon resources to display an icon for the input method in the Keyboard menu: `'kcs#'`, `'kcs4'`, and `'kcs8'`. The resource ID number of the keyboard icon resources must equal the script code of the script system that the input method supports. If your input method supports more than one script system, you can have more than one icon suite, each with the appropriate resource IDs.

Figure 7-10 shows a Keyboard menu displaying a Japanese input method and a Korean input method, as well as keyboard layouts from several other script systems. The Japanese input method is active; its icon is checked in the menu and appears highlighted on the menu bar.

Figure 7-10 Input method icons in the Keyboard menu and menu bar

The pencil icon between the Keyboard menu and the Help menu in Figure 7-10 is the title for the menu belonging to the active input method.

For more information on keyboard icon suites, see the appendix "Keyboard Resources" in this book. For information on script codes, see the chapter "Script Manager" in this book.

Responding to Calls

When a client application makes certain calls to the Text Services Manager application interface, the Text Services Manager in turn calls your text service component. Your text service component responds to these calls by initiating or closing a text service, manipulating text service windows, responding to events or menu commands, and confirming text input.

Initiating a Text Service

When your text service component receives the `InitiateTextService` call, it can commence its operations to provide its text service. That may include opening windows or palettes, initializing data structures, communicating with the user or application, or otherwise getting started with its tasks.

The Text Services Manager may call `InitiateTextService` on its own or in response to receiving the application-interface call `OpenTextService`.

Activating Text Service Component Windows

If a window associated with a TSM document associated with your text service is being activated, your text service component receives the `ActivateTextService` call. You should show floating windows associated with your component instance and prepare to receive and handle events.

If the window is being deactivated, your text service component receives the `DeactivateTextService` call. You should perform any necessary cleanup or other tasks associated with deactivating your current component instance. If your text service component is not an input method, you should also hide all floating windows associated with the document being deactivated. If your text service component is an input method and if the newly activated document does *not* use your text services, you will receive the `HidePaletteWindows` call. At that point you should hide all floating windows associated with the component instance being deactivated.

The Text Services Manager calls `ActivateTextService` and `DeactivateTextService` in response to receiving the application-interface calls `ActivateTSMDocument` and `DeactivateTSMDocument`, respectively.

Responding to Events and Updating the Cursor and Menu

The Text Services Manager (or a client application) is responsible for adding your text service component's menu to the menu bar. When your text service component receives a `GetTextServiceMenu` call, it needs to return a menu handle. The section "Providing Menus and Icons" on page 7-38 gives instructions for creating text service menus and icons.

When your text service component receives the call `TextServiceEvent`, `TextServiceMenuSelect`, or `SetTextServiceCursor`, it should handle the event, menu command, or cursor-drawing if appropriate. For example, when the user enters text, you receive and handle the key-down events; you in turn inform the application what characters to draw in the active input area. When the user makes a menu selection, you are given an opportunity to check whether it is from your menu and then to act on it. You are regularly given the opportunity to redraw the cursor, in case it may be over an area under your control (such as a palette window or the active input area).

The Text Services Manager may call `GetTextServiceMenu` on its own or in response to receiving the application-interface call `OpenTextService`. The Text Services Manager calls `TextServiceEvent`, `TextServiceMenuSelect`, and `SetTextServiceCursor` in response to receiving the application-interface calls `TSMEvent`, `TSMMenuSelect`, and `SetTSMCursor`, respectively.

Confirming Active Text Input

A client application may need your input method text service component to terminate input immediately and confirm any text currently in the active input area. When your text service component receives the call `FixTextService`, it should confirm all text in the active input area, just as if the user had pressed Return. It should send the confirmed text to the client application through the Update Active Input Area Apple event.

The Text Services Manager calls `FixTextService` in response to receiving the application-interface call `FixTSMDocument`.

Closing a Text Service

When your text service is no longer needed, the Text Services Manager calls your text service component's `TerminateTextService` function before calling the Component Manager to close the component. Your text service component should use this time to confirm any active input in progress and then dispose of memory as needed.

The Text Services Manager may call `TerminateTextService` on its own or in response to receiving the application-interface call `CloseTextService`.

Identifying the Supported Scripts and Languages

The Operating System, the Text Services Manager, or a client application may need to determine which scripts and languages your text service component supports. When you receive the `GetScriptLanguageSupport` call, you return that information in a script-language support record.

The `GetScriptLanguageSupport` function and several Text Services Manager application-interface routines use the **script-language record**—and the **script-language support record**—to pass information about the scripts and languages associated with text service components.

The script-language record provides a script code and language code for the script system and the language associated with a given text service component. The script-language record is defined by the `ScriptLanguageRecord` data type as follows:

```
TYPE ScriptLanguageRecord =
   RECORD
      fScript:    ScriptCode;
      fLanguage:  LangCode;
   END;
```

Field descriptions

`fScript` The number that identifies a script system supported by the text service component

`fLanguage` The number that identifies a language associated with the script supported by the text service component

For a list of constants for all defined script and language codes, see the chapter "Script Manager" in this book.

The script-language support record consists of an array of script-language records. It is defined by the `ScriptLanguageSupport` data type as follows:

```
TYPE ScriptLanguageSupport =
   RECORD
      fScriptLanguageCount:   Integer;
      fScriptLanguageArray:   ARRAY[0..0] of ScriptLanguageRecord;
   END;
```

Field descriptions

`fScriptLanguageCount`
The number of script-language records in this script-language support record.

`fScriptLanguageArray`
A variable-length array of script-language records.

The Text Services Manager can call `GetScriptLanguageSupport` on its own or in response to receiving the application-interface call `GetTextServiceLanguage`.

Listing 7-10 gives an example of the response to the `GetScriptLanguageSupport` call by a Chinese input method.

Listing 7-10 Determining the script and language for a text service component

```
TYPE
   scriptHandlePtr= ^ScriptLanguageSupportHandle;

VAR
   scriptHdlPtr:scriptHandlePtr;

{The following is part of the case statement that dispatches }
{ text service component routines. It is the component's }
{ response to receiving a GetScriptLanguageSupport call}

kCMGetScriptLangSupport:
   BEGIN
      scriptHdlPtr := (scriptHandlePtr) @(cmParams^.params[0]);
      IF scriptHdlPtr^ = NIL THEN
```

```
        scriptHdlPtr^ := (ScriptLanguageSupportHandle)NewHandle
                                 (sizeof(ScriptLanguageSupport));
     IF scriptHdlPtr^ <> NIL THEN
           WITH scriptHdlPtr^^^ DO BEGIN
              fScriptLanguageCount := 1;
              fScriptLanguageArray[0].fScript:= smTradChinese;
              fScriptLanguageArray[0].fLanguage:=
                                                 langTradChinese;
              result := noErr;
           END;
     ELSE
        result := memFullErr;
  END;
```

Making Calls

Your text service component needs to make two kinds of calls to the Text Services Manager: calls that cause the sending of an Apple event to a client application, and calls that request a floating window from the Text Services Manager.

Sending Apple Events to Client Applications

Apple events allow text service components to send information to and request specific services of client applications. It is the responsibility of the client application to install Apple event handlers for these Apple events. Using these events, the text service component controls the text services environment by requesting a variety of services from the client application.

Your text service component can send Apple events to request that a client application perform the following actions:

- create or update text in an active input area
- help you track cursor movements by converting global coordinates to the byte offset of characters in the active input area
- help you position items on the screen by converting the byte offset of characters in the active input area to global coordinates
- show or hide a floating input window

Note

Your text service component must always use the `kCurrentProcess` constant as the target address when it creates an Apple event to send to the Text Services Manager. ◆

To send Apple events to a client application, your text service component calls the Text Services Manager `SendAEFromTSMComponent` function. The Text Services Manager then completes the Apple event and sends it to the application. For general information on constructing and sending Apple events, see the discussion of the Apple Event Manager in *Inside Macintosh: Interapplication Communication*.

Listing 7-11 shows an example of a text service component preparing and sending an Update Active Input Area Apple event. The component creates the Apple event and constructs the required parameters, including the text to be sent to the application. It also constructs the optional parameters that specify highlighting and update ranges in the text. It then calls `SendAEFromTSMComponent` to send the Apple event. In this listing, `globalHandle` is a handle to a data structure in which the text service component maintains all information about the text in the active input area.

Listing 7-11 Constructing and sending an Update Active Input Area Apple event

```
FUNCTION MyCreateUpdateInlineAreaAE(globalHandle: TglobalHandle)
: OSErr;
VAR
   psnRecord:              ProcessSerialNumber;
   myErr:                  OSErr;
   addrDescriptor:         AEAddressDesc;
   theAEvent:              AppleEvent;
   theReply:               AppleEvent;
   slRecord:               ScriptLanguageRecord;
   theRangeTableSize:      LongInt;
   theTextData:            Handle;
   theUpdateRangeTable:    TextRangeArray;
   theHiliteRangeTable:    TextRangeArray;
BEGIN
   {Apple event must go to the current process }
   psnRecord.highLongOfPSN := 0;
   psnRecord.lowLongOfPSN  := kCurrentProcess;
   myErr := AECreateDesc(typeProcessSerialNumber, @psnRecord,
                         sizeof(psnRecord), addrDescriptor);
   IF myErr <> noErr THEN
   BEGIN
      MyCreateUpdateInlineAreaAE := myErr;
      Exit(MyCreateUpdateInlineAreaAE);
   END;
```

```
{create the Apple event record}
myErr := AECreateAppleEvent(kTextServiceClass,
                            kUpdateActiveInputArea,
                            addrDescriptor,
                            kAutoGenerateReturnID,
                            kAnyTransactionID, theAEvent);
IF myErr <> noErr THEN
BEGIN
   MyCreateUpdateInlineAreaAE := myErr;
   myErr := AEDisposeDesc(addrDescriptor);   {ignore the error}
   Exit(MyCreateUpdateInlineAreaAE);
END;

{construct the required parameter keyAEServerInstance-- }
{ globalHandle^^.fSelf = global containing component instance}
myErr := AEPutParamPtr(theAEvent, keyAEServerInstance,
                       typeComponentInstance,
                       @globalHandle^^.fSelf,
                       sizeof(ComponentInstance));
IF myErr = noErr THEN
BEGIN
   {construct required parameter keyAEScriptTag }
   { --Korean in this case}
   slRecord.fScript := smKorean;
   slRecord.fLanguage := langKorean;
   myErr := AEPutParamPtr(theAEvent, keyAEScriptTag,
                          typeIntlWritingCode,
                          @slRecord, sizeof(slRecord));
END;
IF myErr = noErr THEN
BEGIN
   {construct required parameter keyAETheData. globalHandle }
   { is a handle to component's data structure describing }
   { all text in the active inline area}
   theTextData := globalHandle^^.fTextData;
   HLock(theTextData);
   myErr := AEPutParamPtr(theAEvent, keyAETheData, typeChar,
                       theTextData^,
                       globalHandle^^.fTextDataLength);
   HUnlock(theTextData);
END;
```

```
   IF myErr = noErr THEN
      {construct the required parameter keyAEFixLength}
      myErr := AEPutParamPtr(theAEvent, keyAEFixLength,
                             typeInteger,
                             @globalHandle^^.fFixedLength,
                             sizeof(LongInt));

   IF myErr = noErr THEN
   BEGIN
      {construct the optional parameter UpdateRangeTable}
      theUpdateRangeTable := globalHandle^^.fUpdateRangeTable;
      theRangeTableSize := sizeof(TextRangeArray)
                             + theUpdateRangeTable.fNumOfRanges
                             * sizeof(TextRange);
      myErr := AEPutParamPtr(theAEvent, keyAEFixLength,
                             typeInteger,
                             @theUpdateRangeTable,
                             theRangeTableSize);
   END;
   IF myErr = noErr THEN
   BEGIN
      {construct the optional parameter HiliteRangeTable}
      theHiliteRangeTable := globalHandle^^.fHiliteRangeTable;
      theRangeTableSize := sizeof(TextRangeArray)
                             + theHiliteRangeTable.fNumOfRanges
                             * sizeof(TextRange);
      myErr := AEPutParamPtr(theAEvent, keyAEFixLength,
                             typeInteger,
                             @theHiliteRangeTable,
                             theRangeTableSize)
   END;
   IF myErr <> noErr THEN
   BEGIN
      MyCreateUpdateInlineAreaAE := myErr;
      myErr := AEDisposeDesc(theAEvent);     {ignore the errors}
      myErr := AEDisposeDesc(addrDescriptor);
      Exit(MyCreateUpdateInlineAreaAE);
   END;

   {send the Apple event}
   myErr := SendAEFromTSMComponent(theAEvent, theReply,
                                kAEWaitReply + kAENeverInteract,
                                kAENormalPriority, 120, NIL, NIL);
   MyCreateUpdateInlineAreaAE := myErr;
```

```
   myErr := AEDisposeDesc(theAEvent);        {ignore the errors}
   myErr := AEDisposeDesc(addrDescriptor);
   myErr := AEDisposeDesc(theReply);
END;
```

Opening Floating Utility Windows

To open a floating utility window in front of the current client application, you use the `NewServiceWindow` function. If the call is successful, `NewServiceWindow` allocates a floating window in the floating window service layer, and returns a pointer to the window. See "Floating Input Windows" on page 7-13 and "Floating Utility Windows" on page 7-14 for a discussion of the Text Services Manager floating window service and the floating window service layer.

Your text service component can open multiple floating windows. When your component receives an event, you must determine if the event belongs to one of your text service floating windows. To get a pointer to the frontmost window in the floating window service layer, call the `GetFrontServiceWindow` function. To find out which part of your floating window an event occurred in, call the `FindServiceWindow` function. Your text service component can close the floating window you originally allocated by using the `CloseServiceWindow` function.

Text Services Manager Reference

This section describes four categories of routines and handlers, and their related constants and data structures:

- Text Services Manager routines called by client applications (the application interface to the Text Services Manager)
- application-supplied handlers for Apple events initiated by text service components
- Text Services Manager routines called by text service components (the component interface to the Text Services Manager)
- text service component routines, called by the Text Services Manager and possibly by client applications

Text Services Manager Routines for Client Applications

The Text Services Manager provides an application interface that allows client applications to use text service components independently of any specific knowledge of those components. Your client application makes these application-level calls to the Text Services Manager, which in turn calls the text service component using the component-level routines described in the section "Text Service Component Routines" on page 7-84.

The routines in the application interface let you

- initialize and close your TSM-aware application
- use TSM documents
- pass events, menu items, and cursor control to text service components
- confirm active input in TSM documents that use input methods
- provide text services to the user
- request a floating input window instead of inline input
- associate scripts and languages with text service components

Initializing and Closing as a TSM-Aware Application

If your client application uses any of the application-level Text Services Manager routines, call the `InitTSMAwareApplication` function immediately after you have called the other Toolbox initialization routines.

The Text Services Manager needs to perform some housekeeping when your client application is closed. To expedite this process, call the `CloseTSMAwareApplication` function when you quit.

InitTSMAwareApplication

The `InitTSMAwareApplication` function informs the Text Services Manager that your application is TSM-aware.

```
FUNCTION InitTSMAwareApplication: OSErr;
```

DESCRIPTION

The Text Services Manager notes that your application is TSM-aware by allocating the necessary data in its internal data structures.

RESULT CODES

`noErr`	No error
`memFullErr`	Insufficient memory to initialize
`tsmAlreadyRegisteredErr`	Application is already TSM-initialized
`tsmNotAnAppErr`	The caller is not an application

SEE ALSO

For sample code that uses the `InitTSMAwareApplication` function, see Listing 7-1 on page 7-18.

CloseTSMAwareApplication

The `CloseTSMAwareApplication` function informs the Text Services Manager that you have closed your application.

```
FUNCTION CloseTSMAwareApplication: OSErr;
```

DESCRIPTION

The Text Services Manager performs necessary housekeeping when your application closes.

Before you call the `CloseTSMAwareApplication` function, be sure that your application disposes of all open TSM documents by calling the `DeleteTSMDocument` function (see page 7-53).

RESULT CODES

`noErr`	No error
`tsmNeverRegisteredErr`	Application was never TSM-initialized

SEE ALSO

For sample code that uses the `CloseTSMAwareApplication` function, see Listing 7-6 on page 7-24.

Creating and Activating TSM Documents

This section describes the functions that let you create, activate, deactivate, and dispose of a TSM document (for details on the contents of a TSM document, see the section "Creating a TSM Document" on page 7-18).

NewTSMDocument

The `NewTSMDocument` function creates a TSM document and returns a handle to the document's ID.

```
FUNCTION NewTSMDocument (numOfInterface: Integer;
                         VAR supportedInterfaceTypes:
                         InterfaceTypeList;
                         VAR idocID: TSMDocumentID;
                         refCon: LongInt): OSErr;
```

`numOfInterface`
The number of supported text service interface types. Currently, this number must be 1.

`supportedInterfaceTypes`
A list of supported interface types. This list helps the Text Services Manager to locate the text services that have the correct interface type. Currently, the Text Services Manager has defined one interface type: `kTextService` (= `'tsvc'`). The data type `InterfaceTypeList` is a simple array of 4-character (`OSType`) tags.

`idocID` Upon successful completion of the call, contains the document identification number of the TSM document created.

`refCon` A reference constant to store in the TSM document record. It may have any value you wish.

DESCRIPTION

Each time your client application calls the `NewTSMDocument` function, the Text Services Manager creates an internal record called a TSM document and returns its ID.

If the call is successful, `NewTSMDocument` opens the default input method text service component of the current keyboard script and assigns it to this document. If `NewTSMDocument` returns `tsmScriptHasNoIMErr`, it has still created a valid TSM document, but has not associated an input method with it.

If `NewTSMDocument` fails to create a new TSM document, it returns an error and sets `idocID` to `NIL`.

RESULT CODES

`noErr`	No error
`memFullErr`	Insufficient memory to open document
`tsmUnsupportedTypeErr`	Supported type was not `'tsvc'`
`tsmNeverRegisteredErr`	Application is not TSM-aware
`tsmScriptHasNoIMErr`	Current script does not use input methods
`tsmCantOpenComponentErr`	Cannot open default input of current script

SEE ALSO

For sample code that uses the `NewTSMDocument` function, see Listing 7-2 on page 7-19.

ActivateTSMDocument

The `ActivateTSMDocument` function instructs the Text Services Manager to mark the TSM document associated with a newly active window as active.

```
FUNCTION ActivateTSMDocument (idocID: TSMDocumentID): OSErr;
```

`idocID` A TSM document identification number created by a prior call to the `NewTSMDocument` function (see page 7-50).

DESCRIPTION

When a window that has an associated TSM document becomes active, your client application must call the `ActivateTSMDocument` function to inform the Text Services Manager that the document is activated and is ready to use text service components.

`ActivateTSMDocument` calls the equivalent text service component routine `ActivateTextService` (see page 7-85) for all open text service components associated with the TSM document.

If a text service component has a menu, the Text Services Manager inserts the menu into the menu bar.

RESULT CODES

`noErr`	No error
`tsmInvalidDocIDErr`	Document is not a valid TSM document

SEE ALSO

For sample code that uses the `ActivateTSMDocument` function, see Listing 7-3 on page 7-21.

DeactivateTSMDocument

The `DeactivateTSMDocument` function instructs the Text Services Manager to mark the TSM document as inactive.

```
FUNCTION DeactivateTSMDocument (idocID: TSMDocumentID): OSErr;
```

`idocID` A TSM document identification number created by a prior call to the `NewTSMDocument` function (see page 7-50).

DESCRIPTION

The `DeactivateTSMDocument` function lets you inform the Text Services Manager that a TSM document in your client application is no longer active and must temporarily stop using text service components.

The Text Services Manager calls the equivalent text service component function `DeactivateTextService` (see page 7-85) for any text service component associated with the TSM document being deactivated.

IMPORTANT

Once your application is initialized as a TSM-aware application, at least one TSM document must always be active when your application is active. If a situation arises in which you are a TSM-aware application but all of your TSM documents are inactive, any text service component that has a menu or palette windows will be unable to communicate with the user. The best policy is to always create a TSM document, even if only a dummy document, immediately after initializing as a TSM-aware application. ▲

RESULT CODES

`noErr`	No error
`tsmInvalidDocIDErr`	Document is not a valid TSM document

SEE ALSO

For sample code that uses the `DeactivateTSMDocument` function, see Listing 7-3 on page 7-21.

DeleteTSMDocument

The `DeleteTSMDocument` function closes all opened text service components for the TSM document.

```
FUNCTION DeleteTSMDocument (idocID: TSMDocumentID): OSErr;
```

`idocID` A TSM document identification number created by a prior call to the `NewTSMDocument` function (see page 7-50).

DESCRIPTION

When your application disposes of a TSM document, it must call the `DeleteTSMDocument` function to inform the Text Services Manager that the document is no longer using text service components. `DeleteTSMDocument` invokes the Component Manager `CloseComponent` function for each open text service component associated with this document. It also disposes of the internal data structure for the TSM document.

RESULT CODES

`noErr`	No error
`tsmInvalidDocIDErr`	Document is not a valid TSM document
`tsmNeverRegisteredErr`	Application is not TSM-aware

Passing Events to Text Service Components

This section describes a function that lets you instruct the Text Services Manager to pass certain events to the appropriate text service component.

TSMEvent

The `TSMEvent` function passes all events obtained from the `WaitNextEvent` function, including null events, to the Text Services Manager.

```
FUNCTION TSMEvent (VAR event: EventRecord): Boolean;
```

`event` The event record for the event that has been obtained from `WaitNextEvent`.

DESCRIPTION

Your client application regularly obtains events such as key-down events from the Toolbox Event Manager function `WaitNextEvent`. Some of these events may need to be handled by text service components. The `TSMEvent` function lets you pass those events to the Text Services Manager. The Text Services Manager dispatches the passed events to the appropriate text service components by calling the `TextServiceEvent` function for each component (see page 7-87).

If `TSMEvent` returns `FALSE`, you need to process the event as you normally do. If `TSMEvent` returns `TRUE`, the event has been handled by a text service component and is now a null event. You should process the null event as you normally do.

Note
The way the Text Services Manager uses and dispatches Apple events creates the potential for a reentrance situation that your client application should know about and be prepared to handle. When your application calls `TSMEvent`, the Text Services Manager uses the Apple Event Manager function `AESend` to pass data to your application through an Apple event. Your Apple event handler is thus invoked before the `TSMEvent` trap has returned. ◆

SEE ALSO

The `WaitNextEvent` function is described in the Event Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials.*

For sample code that uses the `TSMEvent` function, see Listing 7-4 on page 7-22.

Passing Menu Selections and Cursor Setting

This section describes two functions, `TSMMenuSelect` and `SetTSMCursor`, that let you instruct the Text Services Manager to pass menu commands and cursor control to the appropriate text service component.

TSMMenuSelect

The `TSMMenuSelect` function gives the specified text service component a chance to reply to a menu selection.

```
FUNCTION TSMMenuSelect (menuResult: LongInt): Boolean;
```

`menuResult`
The result from the Menu Manager `MenuSelect` function.

DESCRIPTION

When the user chooses a menu item, the item may belong to a text service component's menu. To provide an opportunity for the text service component to reply to its menu selections, your application should call `TSMMenuSelect` with the result from the Menu Manager `MenuSelect` function.

`TSMMenuSelect` returns `FALSE` if a text service component did not handle the menu selection. In this case, your client application should process the menu selection normally. `TSMMenuSelect` returns `TRUE` when a text service component handled the menu selection. In this case, you should take no action.

After `TSMMenuSelect` returns, your application should—as usual—call the Menu Manager function `HiliteMenu` with the `menuID` parameter set to 0 to remove the highlighting from the menu title.

SEE ALSO

The Menu Manager is described in *Inside Macintosh: Macintosh Toolbox Essentials*.

SetTSMCursor

The `SetTSMCursor` function provides an opportunity for the text service component to set the shape of the cursor. If the text service component does not respond, your application may set the cursor.

```
FUNCTION SetTSMCursor (mousePos: Point): Boolean;
```

`mousePos` A QuickDraw point indicating the position (in global coordinates) of the cursor in your application.

DESCRIPTION

Your client application is responsible for setting the cursor to an appropriate shape as it passes over your various user interface elements. It is also necessary to provide an opportunity for a text service component to set the cursor over its own user interface elements. The `SetTSMCursor` function allows the text service component to control the shape of the cursor if appropriate.

Call `SetTSMCursor` whenever you would normally call the QuickDraw `SetCursor` procedure. When `SetTSMCursor` returns `TRUE`, the cursor is positioned in a text service component window or in the active input area and it has been set by a text service component. Your client application should not set the cursor in this case. When `SetTSMCursor` returns `FALSE`, the cursor has not been set, and your client application may set it.

`SetTSMCursor` calls the equivalent text service component function `SetTextServiceCursor` (page 7-88) for each open text service component to provide an opportunity for each one to set shape of the cursor. If a text service component actually changes the shape of the cursor, the Text Services Manager does not call `SetTextServiceCursor` for the rest of the text service components and returns `TRUE`. If none of the text service components sets the cursor, then `SetTSMCursor` returns `FALSE`.

SEE ALSO

The `SetCursor` procedure is described in the QuickDraw chapters of *Inside Macintosh: Imaging*.

Confirming Active Input in a TSM Document

This section describes the `FixTSMDocument` function, which allows you to explicitly confirm text in the active input area.

FixTSMDocument

The `FixTSMDocument` function informs the Text Services Manager that input in the active input area of a specified TSM document has been interrupted, and that the text service component must confirm the text and terminate user input.

```
FUNCTION FixTSMDocument (idocID: TSMDocumentID): OSErr;
```

`idocID` The identification number of a TSM document created by a prior call to the `NewTSMDocument` function (see page 7-50).

DESCRIPTION

Typically, an inline input text service component removes confirmed input from the active input area each time the user presses the Return key, and passes the confirmed text to your application through an Apple event.

In certain situations, however, your client application may need to inform the text service component that there has been an interruption in user input for a specific TSM document. In this case you call the `FixTSMDocument` function to give the input method text service component the opportunity to confirm any input in progress.

For instance, if the user clicks in the close box of the window in which active input is taking place, call `FixTSMDocument` before you close the window. The text service component will pass you the current contents (both converted and unconverted) of the active input area as confirmed text.

For simple activating and deactivating of your application's window, it is not necessary to confirm the text in the active inline area. The input method saves the text and restores it when your window is reactivated.

RESULT CODES

`noErr`	No error
`tsmInvalidDocIDErr`	The document is not a valid TSM document
`tsmDocNotActiveErr`	The TSM document is not active
`tsmTSNotOpenErr`	The default input method is not open

SEE ALSO

For sample code that uses the `FixTSMDocument` function, see Listing 7-5 on page 7-23.

Making Text Services Available to the User

This section describes functions that let you provide ways for the user to choose, open, and close text service components that are not input methods.

Your client application is responsible for providing a way—usually a menu—for the user to choose from among all available text service components. To get a list of available text service components to display in a menu, call the `GetServiceList` function. Be sure to filter out input methods, because the Keyboard menu already displays them.

When the user chooses a text service component that is not an input method, call the `OpenTextService` function to add the text service component to the TSM document. The `OpenTextService` and `CloseTextService` functions let you inform the Text Services Manager that a user of your client application has chosen to open or close a text service component. The Text Services Manager then opens or closes the component and associates it with a TSM document or ends the association as appropriate.

GetServiceList

The `GetServiceList` function obtains a complete list of text service components of a given kind available to the user of your client application.

```
FUNCTION GetServiceList (numOfInterfaceTypes: Integer;
                         supportedInterfaceTypes:
                         InterfaceTypeList;
                         VAR serviceInfo: TextServiceListHandle;
                         VAR seedValue: LongInt): OSErr;
```

`numOfInterfaceTypes`
The number of interface types supported by your client application.

`supportedInterfaceTypes`
A list of the interface types supported by your client application. The data type `InterfaceTypeList` is a simple list of 4-character (`OSType`) tags.

`serviceInfo`
A handle to the text service component list data structure. If the handle is `NIL`, the Text Services Manager allocates the handle; otherwise, it assumes the handle is a valid text service component list handle, as defined by the `TextServiceListHandle` data type.

`seedValue` A value that indicates whether the list of text service components returned by `GetServiceList` may have been modified. This value is returned in this parameter after the Text Services Manager calls the Component Manager `GetComponentListModSeed` function.

DESCRIPTION

When your client application calls `GetServiceList`, the Text Services Manager locates all the text service components that support the specified interface and text service component types and creates a text service component list, defined by the `TextServiceList` data type, that contains an entry for each of the text service components.

It is possible to register text service components or withdraw them from registration at any time. Once it has compiled a list of text services, the Text Services Manager invokes the `GetComponentListModSeed` function and returns the value in the `modseed` parameter. You can save that value and, the next time you need to draw or regenerate the list of services, call the Component Manager `GetComponentListModSeed` function. If the seed value differs from the one you received from your last call to `GetServiceList`, you need to call `GetServiceList` once more to update the information. Alternatively, you can simply call `GetServiceList` each time you need to update the list, although that may be less efficient.

`GetServiceList` uses the text service component information record, defined by the `TextServiceInfo` data type, and the text service component list record, defined by the `TextServiceList` data type.

```
TYPE TextServiceInfo =
   RECORD
      fComponent: Component;
      fItemName:  Str255;
   END;
   TextServicesInfoPtr = ^TextServiceInfo;
```

Field descriptions

`fComponent` A component identifier for this text service component. You can use the component identifier in Text Services Manager functions that open or obtain information about a text service component.

`itemName` A Pascal string with the name of a text service component. (The script system to use for displaying the string is specified in the `componentFlags` field of the component description record. See page 7-15.)

```
TYPE TextServiceList =
   RECORD
      fTextServiceCount:   Integer;
      fServices:           ARRAY[0..0] of TextServiceInfo;
   END;
   TextServiceListPtr = ^TextServiceList;
   TextServiceListHandle = ^TextServiceListPtr;
```

Field descriptions

`fTextServiceCount` An integer that provides the number of text service components in the text service component list.

`fServices` A variable-length array of text service component information records.

RESULT CODES

`noErr`	No error
`memFullErr`	Insufficient memory
`tsmUnsupportedTypeErr`	Supported type was not `'tsvc'`

OpenTextService

The `OpenTextService` function instructs the Text Services Manager to open a text service component that a user has chosen and to associate it with a TSM document.

```
FUNCTION OpenTextService (idocID: TSMDocumentID;
                          aComponent: Component;
                          VAR aComponentInstance:
                          ComponentInstance): OSErr;
```

`idocID` The identification number of a TSM document created by a prior call to the `NewTSMDocument` function (see page 7-50).

`aComponent`
A component identifier for this text service component.

`aComponentInstance`
Upon completion of the call, contains a component instance. This value identifies your application's connection to a text service component. You must supply this value whenever you call the text service functions provided by the component directly.

DESCRIPTION

You can obtain the component identifier to pass in `aComponent` by comparing the menu item name selected by the user with the component item names in the `TextServiceList` record obtained by calling `GetServiceList`.

The Text Services Manager opens the requested component by calling the Component Manager `OpenComponent` function.

If the specified text service component is already open, the Text Services Manager does not open it again and the `tsmComponentAlreadyOpenErr` error message is returned as a result code. Whether or not the text service is open, the Text Services Manager calls the functions `InitiateTextService` (see page 7-84) and `ActivateTextService` (see page 7-85) for the given text service and returns a valid component instance. Upon completion of the `OpenTextService` call, the selected text service component is initialized and active.

Note
This function is for opening text service components other than input methods. Your application does not need to open or close input methods. ◆

RESULT CODES

`noErr`	No error
`tsmInvalidDocIDErr`	The document is not a valid TSM document
`tsmComponentAlreadyOpenErr`	Component is already open for this document
`tsmCantOpenComponentErr`	Component doesn't exist or won't open

CloseTextService

The `CloseTextService` function deactivates the active TSM document's association with the specified text service and closes the service component.

```
FUNCTION CloseTextService (idocID: TSMDocumentID;
                           aComponentInstance: ComponentInstance):
                           OSErr;
```

`idocID` The identification number of a TSM document created by a prior call to the `NewTSMDocument` function (see page 7-50).

`aComponentInstance`
The component instance created by a prior call to `OpenTextService`.

DESCRIPTION

When a user wants to close an opened text service component, your client application should call `CloseTextService`.

If the text service component displays a menu, the Text Services Manager removes the menu from the menu bar.

Note
This function is for closing text service components other than input methods. Your application does not need to open or close input methods. ◆

RESULT CODES

`noErr`	No error
`tsmInvalidDocIDErr`	The document is not a valid TSM document
`tsmNoOpenTSErr`	The component for this document is not open

Requesting a Floating Input Window

In certain situations, bottomline input with a floating input window is preferable to inline input for text input users. The Text Services Manager provides two ways to control how the floating input window is used: with a single specified TSM document or with all documents of a given application.

UseInputWindow

The `UseInputWindow` function associates a floating input window with a particular TSM document or with all TSM documents of an application.

```
FUNCTION UseInputWindow (idocID: TSMDocumentID;
                         useWindow: Boolean): OSErr;
```

`idocID` The TSM document ID of the particular TSM document to be associated with the floating input window. If `NIL`, this call affects all your application's TSM documents.

`useWindow` A Boolean value that indicates whether to use the floating input window. Set it to `TRUE` if you want to use a floating window; set it to `FALSE` if you do not want to use a floating window.

DESCRIPTION

The Text Services Manager provides a floating input window for your application's use if you call `UseInputWindow` with a value of `TRUE` in the `useWindow` parameter. To specify inline input instead, call `UseInputWindow` with a value of `FALSE` in the `useWindow` parameter.

The default value for `useWindow` is `FALSE`; if you do not call `UseInputWindow`, the Text Services Manager assumes that your application wants to use inline input. If your application wants to save the user's choice, it can put the last-used value for `useWindow` in a preferences file before quitting.

If you pass a valid TSM document ID for the `idocID` parameter, the `useWindow` parameter affects only that TSM document. If you pass `NIL` for the `idocID` parameter, the `useWindow` parameter affects all your application's TSM documents, including documents you create after making this call.

RESULT CODES

`noErr`	No error
`tsmInvalidDocIDErr`	The document is not a valid TSM document
`tsmNeverRegisteredErr`	Application is not TSM-aware

Associating Scripts and Languages With Components

The utility routines described in this section allow you to

- assign a particular text service component as the default component to be associated with a given script system and language
- determine which text service component is the default component associated with a given script system and language

- determine the script system and language combination for the currently active text service component
- assign a script system and language combination to the currently active text service component

In addition to these routines, you can use the text service component function `GetScriptLanguageSupport` (described on page 7-90) to determine which additional scripts and languages a text service component supports.

These routines make use of the script-language record, described under "Identifying the Supported Scripts and Languages" on page 7-42.

SetDefaultInputMethod

The operating system uses the `SetDefaultInputMethod` function to assign a default (input method) text service component to a given script and language.

```
FUNCTION SetDefaultInputMethod (ts: Component;
                                VAR slRecord: ScriptLanguageRecord):
                                OSErr;
```

`ts`	The component identifier of the input method text service component to be associated with the script and language combination given in the `slRecord` parameter.
`slRecord`	A script-language record that describes the script and language combination to be associated with the input method text service component specified in the `ts` parameter.

DESCRIPTION

The operating system uses `SetDefaultInputMethod` to associate an input method text service component with a given script and language. The operating system calls this function when the user expresses input method preferences through the Keyboard menu, Keyboard control panel, or other device. The associations made with this function are permanent; that is, they persist after restart.

If the script code and language code specified in the script-language record are incompatible, `SetDefaultInputMethod` returns the error `paramErr`.

RESULT CODES

`noErr`	No error
`paramErr`	The script does not match the language
`tsmScriptHasNoIMErr`	Current script does not use input methods
`tsmCantOpenComponentErr`	Cannot open default input of current script

GetDefaultInputMethod

The `GetDefaultInputMethod` function returns the default (input method) text service component for a given script and language.

```
FUNCTION GetDefaultInputMethod (VAR ts: Component;
                                VAR slRecord: ScriptLanguageRecord):
                                OSErr;
```

`ts` The component identifier of the input method text service component that is associated with the script and language combination given in the `slRecord` parameter.

`slRecord` A script-language record that describes the script and language combination that is associated with the input method text service specified in the `ts` parameter.

DESCRIPTION

The operating system uses `GetDefaultInputMethod` to find out which input method to activate when the user selects a new keyboard script from the Keyboard menu or by Command-key combination, or when an application calls `KeyScript` to change keyboard scripts.

In versions of Japanese system software starting with KanjiTalk 7.0, if the default input method is an old (pre-KanjiTalk 7.0) non-TSM-aware method, `GetDefaultInputMethod` returns the error `tsmInputMethodIsOldErr`. In that case the `ts` parameter contains the script code of the old input method in its high-order word, and the reference ID of the old input method in its low-order word.

RESULT CODES

`noErr`	No error
`paramErr`	The script does not match the language
`tsmScriptHasNoIMErr`	The script does not use input methods
`tsmInputMethodIsOldErr`	The default input method is old-style

SetTextServiceLanguage

The `SetTextServiceLanguage` function changes the current input script and language.

```
FUNCTION SetTextServiceLanguage (VAR slRecord:
                                 ScriptLanguageRecord): OSErr;
```

`slRecord` A script-language record for the current text service component.

DESCRIPTION

The operating system calls this Text Services Manager function when the user switches the keyboard script, so that the Text Services Manager can synchronize the input method with the current keyboard script.

RESULT CODES

`noErr`	No error
`paramErr`	The script does not match the language
`tsmCantOpenComponentErr`	Cannot open default input of the script

GetTextServiceLanguage

The `GetTextServiceLanguage` function returns the language supported by the default (current) input method text service component for the current keyboard script.

```
FUNCTION GetTextServiceLanguage (VAR slRecord:
                                 ScriptLanguageRecord): OSErr;
```

`slRecord` A script-language record that, upon completion of the call, describes the language supported by the current text service component.

RESULT CODES

`noErr` No error

Apple Event Handlers Supplied by Client Applications

This section describes the Apple events for which client applications must install handlers. Text service components request action from and send information to client applications through these Apple events.

Your application uses these Apple events to receive text from text service components, to show or hide input windows, and to convert screen positions to text offsets—and vice versa—for text service components. The conversion operations are used to track mouse events and determine screen locations of text in the active input area.

The Apple events described in this section are all organized under the `kTextServiceClass` constant with a value of `'tsvc'`.

lists the Apple event ID constants for the Apple events described in this section.

Table 7-1 Apple event ID constants

Constant	Value	Explanation
`kUpdateActiveInputArea`	`'updt'`	Update Active Input Area
`kPos2Offset`	`'p2st'`	Position To Offset
`kOffset2Pos`	`'st2p'`	Offset To Position
`kShowHideInputWindow`	`'shiw'`	Show/Hide Input Window

Table 7-2 shows the Apple event keyword constants used in the Apple events described in this section.

Table 7-2 Apple event keyword constants

Constant	Value	Meaning
`keyAETSMDocumentRefcon`	`'refc'`	TSM document reference constant
`keyAEServerInstance`	`'srvi'`	Component instance
`keyAETheData`	`'kdat'`	Text from active input area
`keyAEScriptTag`	`'sclg'`	Script-language record
`keyAEFixLength`	`'fixl'`	Length of confirmed text
`keyAEHiliteRange`	`'hrng'`	Highlight range in text
`keyAEUpdateRange`	`'udng'`	Update range in text
`keyAEClauseOffsets`	`'clau'`	Clause offsets array
`keyAECurrentPoint`	`'cpos'`	Current point
`keyAEDragging`	`'bool'`	Dragging flag
`keyAEOffset`	`'ofst'`	Byte offset in text
`keyAERegionClass`	`'rgnc'`	Region class
`keyAEPoint`	`'gpos'`	Calculated point
optional keyword for Update Active Input Area		
`keyAEPinRange`	`'pnrg'`	Range for scrolling
optional keywords for Offset To Position		
`keyAETextFont`	`'ktxf'`	Text font
`keyAETextPointSize`	`'ktps'`	Text size
`keyAETextLineHeight`	`'ktlh'`	Text line height
`keyAETextLineAscent`	`'ktas'`	Font ascent

Table 7-2 Apple event keyword constants (continued)

Constant	Value	Meaning
keyAEAngle	'kang'	Text angle
optional keyword for Position To Offset		
keyAELeadingEdge	'klef'	Leading-edge Boolean

Table 7-3 lists the Apple event descriptor types discussed in this section.

Table 7-3 Apple event descriptor types

Constant	Value	Meaning
typeComponentInstance	'cmpi'	Server instance
typeTextRangeArray	'tray'	Text range array
typeOffsetArray	'ofay'	Offset array
typeIntlWritingCode	'intl'	Script-language record
typeQDPoint	'QDpt'	QuickDraw point
typeAEText	'tTXT'	Apple event text
typeText	'TEXT'	Plain text
typeTextRange	'txrn'	A text range record
typeTSMDocumentRefcon	'refc'	TSM document reference constant
typeFixed	'fixd'	Fixed 16.16 format

Table 7-4 lists the Apple event descriptor type constants for region class discussed in this section.

Table 7-4 Apple event descriptor type constants for the Apple event region class

Constant	Value
kTSMOutsideOfBody	1
kTSMInsideOfBody	2
kTSMInsideOfActiveInputArea	3

For the values of standard Apple event constants used in the following section not listed in these tables, see the *Apple Event Registry: Standard Suites*.

Creating and Updating an Active Input Area

The text service component uses the Update Active Input Area Apple event to request that your client application create and update an active input area, and accept confirmed text. For details on active input areas, see "Inline Input" on page 7-11.

Update Active Input Area—Creating and Updating an Active Input Area

Event class	`kTextServiceClass`
Event ID	`kUpdateActiveInputArea`
Requested action	Update a range of text. Specify any necessary highlighting with offsets in the optional `keyAEHiliteRange` parameter.
Required parameters	
Keyword:	`keyAETSMDocumentRefcon`
Descriptor type:	`typeLongInteger`
Data:	A TSM document specifier (reference constant) supplied by the application in a prior call to the `NewTSMDocument` function (see page 7-50). This value is associated with the TSM document whose active input area is to be updated.
Keyword:	`keyAEServerInstance`
Descriptor type:	`typeComponentInstance`
Data:	A component instance value created by a prior call to the Component Manager `OpenComponent` function. This value identifies the text service component.
Keyword:	`keyAETheData`
Descriptor type:	`typeChar`
Data:	Text data that has been processed in some way by a text service component.
Keyword:	`keyAEScriptTag`
Descriptor type:	`typeIntlWritingCode`
Data:	The script code and language code associated with the text returned in the `keyAETheData` parameter. The information is passed in a script-language record, as defined on page 7-42.

Update Active Input Area—Creating and Updating an Active Input Area (continued)

Required parameters

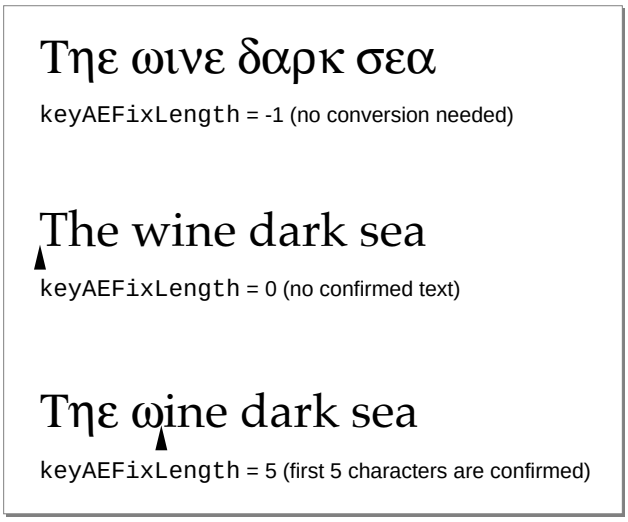
Keyword:	`keyAEFixLength`
Descriptor type:	`typeLongInteger`
Data:	The length of the confirmed text in the active inline area.

If the value of `keyAEFixLength` is –1, the text contained in the `keyAETheData` parameter is to completely replace the current selection in the application window. In this case, there is to be no active input area, the text is all considered to be confirmed, and is to be made part of the body text of the client application.

If the value is 0, an active input area is in process, but there is no completely confirmed text being sent.

If the value is greater than 0, the text specified in the `keyAETheData` parameter up to the indicated offset is confirmed data and should be consumed by the application. The Text Services Manager considers any text beyond the offset specified by the `keyAEFixLength` parameter to be inside the active input area with the starting point of the active input area at that offset. This is illustrated in Figure 7-11.

Figure 7-11 Updating text in an active input area



Update Active Input Area—Creating and Updating an Active Input Area (continued)

Optional parameters

Keyword: `keyAEHiliteRange`

Descriptor type: `typeTextRangeArray`

Data: An array that specifies the ranges of text to be highlighted in the active input area. It also specifies caret position. There are 5 types of highlighting:

Constant	Applies to
`kCaretPosition`	The caret position only
`kRawText`	All of the unconverted text
`kSelectedRawText`	Part of the unconverted text
`kConvertedText`	All of the converted text
`kSelectedConvertedText`	Part of the converted text

For instance, the input method may have the application highlight all raw text with a gray underline; but if it needs to further highlight a selection within that raw text, it may specify a different underline for the selected raw text. The text range array is an array of text-range records, each of which has this form:

```
TYPE TextRange =
  RECORD
    fStart:           LongInt;
    fEnd:             LongInt;
    fHiliteStyle:     Integer;
  END;
```

For the text-range record whose highlight style is `kCaretPosition`, both `fStart` and `fEnd` are the same and denote the position of the caret.

Negative values for a text range mean that the specified range only adds to, rather than replaces, any current highlighting for the specified type of text.

Update Active Input Area—Creating and Updating an Active Input Area (continued)

Optional parameters

Keyword: `keyAEUpdateRange`

Descriptor type: `typeTextRangeArray`

Data: An array of text-range records that indicates the update range of the active input area (in many circumstances, not all of the active input area needs updating). Update Active Input Area always uses the text-range records in the text range array in pairs. The first record (0) specifies a range of old text (text in the inline buffer) to be updated; the second record (1) specifies the range of text in `keyAETheData` that is to replace that old text. In general, the record *n* (*n* >= 0, *n* is an even number) specifies the range of old text to be updated and the record *n* + 1 specifies the range of new text to replace the corresponding old text. (The `fHiliteStyle` field is ignored.)

Keyword: `keyAEPinRange`

Descriptor type: `typeTextRange`

Data: A text range record that specifies a start offset and and an end offset that should be scrolled into view if the text specified by these offsets is not already in view. (The `fHiliteStyle` field is ignored.)

Keyword: `keyAEClauseOffsets`

Descriptor type: `typeOffsetArray`

Data: An offset array (defined by the `OffsetArray` data type) that specifies offsets of word or clause boundaries of the new text. Offsets are from the start of the active input area. Applications can use this information for word selection or other purposes.

```
TYPE OffsetArray =
  RECORD
    fNumOfOffsets: Integer;
    fOffset: ARRAY[0..0] of LongInt;
  END;
```

The `numOfOffsets` field contains an integer that specifies the number of offsets in the offset array. The `fOffset` field is an array of long integers with the number of entries specified in the `numOfOffsets` field.

Return parameter

Keyword: `keyErrorNumber`

Descriptor type: `typeShortInteger`

Data: Any errors that the application needs to return to the text service component. The application must pass Memory Manager, TextEdit, or other errors that it receives through to the component; otherwise, it should pass 0 (`noErr`).

The text range array data structure used in the `keyAEHiliteRange` and `keyAEUpadateRange` parameters described above is defined by the `TextRangeArray` data type:

```
TYPE TextRangeArray =
   RECORD
      fNumOfRanges:  Integer;
      fRange:        ARRAY[0..0] of TextRange;
   END;
```

The `fNumOfRanges` field contains an integer that indicates how many text ranges this array holds. The `fRange` field contains a series of text-range records. (If the array consists of more than one text-range record, the size of the array must be calculated as `fNumOfRanges * SizeOf(fRange)`.)

For sample code that handles the Update Active Input Area Apple event, see Listing 7-7 on page 7-26.

Converting Global Coordinates to Text Offsets

The Position To Offset Apple event requests a client application to convert specified global coordinates to byte offsets in text. The text service component uses this Apple event for mouse tracking, in order to draw the caret, highlight text, or adjust the cursor appearance.

Position To Offset—Converting Global Coordinates to Text Offset

Event class	`kTextServiceClass`
Event ID	`kPos2Offset`
Requested action	Convert global coordinates specified in the `keyAECurrentPoint` parameter to a byte offset. If the click is within the limits of the active input area, the offset is relative to the start of the active input area. Otherwise, the offset is relative to the start of the application's body text. The client application specifies the classification of the location of the offset in the `keyAERegionClass` return parameter.
Required parameters	
Keyword:	`keyAETSMDocumentRefcon`
Descriptor type:	`typeLongInteger`
Data:	A TSM document specifier (reference constant) supplied by the application in a prior call to the `NewTSMDocument` function (see page 7-50). This value is associated with the TSM document affected by this event.

Position To Offset—Converting Global Coordinates to Text Offset (continued)

Required parameters

Keyword: `keyAEServerInstance`

Descriptor type: `typeComponentInstance`

Data: A component instance value created by a prior call to the Component Manager `OpenComponent` function. This value identifies the text service component.

Keyword: `keyAECurrentPoint`

Descriptor type: `typePoint`

Data: A point that contains the global coordinates that describe the current mouse position.

Optional parameter

Keyword: `keyAEdragging`

Descriptor type: `typeBoolean`

Data: A Boolean value that indicates whether the input method is currently tracking the mouse—that is, whether the user is dragging the current selection. If it is `TRUE`, the application should pin the cursor to the limits of the active input area (to avoid highlighting beyond the limits of the active input area).

Return parameters

Keyword: `keyAEOffset`

Descriptor type: `typeLongInteger`

Data: A byte offset that specifies the character corresponding to the current mouse position (`keyAECurrentPoint`). If the click is within the limits of the active input area, the offset is relative to the start of the active input area. Otherwise, the offset is relative to the start of the application's body text.

Keyword: `keyAERegionClass`

Descriptor type: `typeShortInteger`

Data: The classification of the position specified in the `keyAEOffset` parameter. Three constants define the classification:

Constant	Value
`kTSMOutsideOfBody`	1
`kTSMInsideOfBody`	2
`kTSMInsideOfActiveInputArea`	3

A value of `kTSMOutsideOfBody` means that the offset is outside the application's body text. A value of `kTSMInsideOfBody` means that the offset is inside the body text. `kTSMInsideOfActiveInputArea` means that the offset is inside the active input area.

continued

Position To Offset—Converting Global Coordinates to Text Offset (continued)

Return parameters

Keyword:	`keyErrorNumber`
Descriptor type:	`typeShortInteger`
Data:	Any errors that the application needs to return to the text service component. The application must pass Memory Manager, TextEdit, or other errors that it receives through to the component; otherwise, it should pass 0 (`noErr`).

Optional return parameter

Keyword:	`keyAELeadingEdge`
Descriptor type:	`typeBoolean`
Data:	A Boolean value that is equivalent to the `leadingEdge` parameter of the QuickDraw `PixelToChar` function. It is `TRUE` if the specified point corresponds to the leading edge of the character whose offset is returned; it is `FALSE` if the specified point corresponds to the trailing edge of the character.

For sample code that handles the Position To Offset Apple event, see Listing 7-8 on page 7-30.

Converting Text Offsets to Global Coordinates

The Offset To Position Apple event requests that a client application convert byte offsets in text to global coordinates. The text service component uses this Apple event to determine where in the active input area to draw an element (such as the caret or a palette of conversion choices) that relates to a particular character.

Offset To Position—Converting Text Offsets to Global Coordinates

Event class	`kTextServiceClass`
Event ID	`kOffset2Pos`
Requested action	Convert a specified byte offset into global coordinates. The offset value passed to the client application is relative to the start of the active input area. If there is no active input area, the offset is relative to the start of the current text body.

Required parameters

Keyword:	`keyAETSMDocumentRefcon`
Descriptor type:	`typeLongInteger`
Data:	A TSM document specifier (reference constant) supplied by the application in a prior call to the `NewTSMDocument` function (see page 7-50). This value is associated with the TSM document affected by this event.

Offset To Position—Converting Text Offsets to Global Coordinates (continued)

Required parameters

Keyword:	`keyAEServerInstance`
Descriptor type:	`typeComponentInstance`
Data:	A component instance value returned by a prior call to the Component Manager `OpenComponent` function. This value identifies the text service component.
Keyword:	`keyAEOffset`
Descriptor type:	`typeLongInteger`
Data:	The text offset to be converted into a global point. Offset is in terms of bytes from the start of the active input area.

Return parameters

Keyword:	`keyAEPoint`
Descriptor type:	`typePoint`
Data:	A point that contains the global coordinates obtained by converting the byte offset passed in the `keyAEOffset` parameter.
Keyword:	`keyErrorNumber`
Descriptor type:	`typeShortInteger`
Data:	`errOffsetInvalid` indicates that there is no text at the offset. `errOffsetIsOutsideOfView` indicates that the text offset is out of view. The application must pass Memory Manager, TextEdit, or other errors that it receives through to the component; otherwise, it should pass 0 (`noErr`).

Optional return parameters

Keyword:	`keyAETextFont`
Descriptor type:	`typeLongInteger`
Data:	The font of the text in the active input area. The application can send this information to the input method to help the input method position the active input area.
Keyword:	`keyAETextPointSize`
Descriptor type:	`typeFixed`
Data:	The size of the text in the active input area. The application can send this information to the input method to help the input method position the active input area.
Keyword:	`keyAETextLineHeight`
Descriptor type:	`typeShortInteger`
Data:	The line height of the text in the active input area. The application can send this information to the input method to help the input method position the active input area.

continued

Offset To Position—Converting Text Offsets to Global Coordinates (continued)

Optional return parameters

Keyword:	`keyAETextLineAscent`
Descriptor type:	`typeShortInteger`
Data:	The ascent height of the text in the active input area. The application can send this information to the input method to help the input method position the active input area.
Keyword:	`keyAEAngle`
Descriptor type:	`typeFixed`
Data:	The orientation of the text in the active input area. The value 90 specifies a horizontal line direction and 180 specifies a vertical line direction. The application can send this information to the input method to help the input method position the active input area.

For sample code that handles the Offset To Position Apple event, see Listing 7-9 on page 7-33.

Showing or Hiding the Floating Input Window

Input methods that supply floating input windows for bottomline input may need to show or hide the input window at various times. The Show/Hide Input Window Apple event requests the client application to make the floating input window either visible or not visible, so that an input method can offer any of the above options.

Note

If your application is not displaying its own floating input window, you can ignore this Apple event. ◆

Show/Hide Input Window—Showing or Hiding the Floating Input Window

Event class	`kTextServiceClass`
Event ID	`kShowHideInputWindow`
Requested action	Make the bottomline floating input window either visible or not visible, depending on the value of the `keyAEShowHideInputWindow` parameter.
Required parameters	
Keyword:	`keyAETSMDocumentRefcon`
Descriptor type:	`typeLongInteger`
Data:	A TSM document specifier (reference constant) supplied by the application in a prior call to the `NewTSMDocument` function (see page 7-50). This value is associated with the TSM document for the window being shown or hidden.
Keyword:	`keyAEServerInstance`
Descriptor type:	`typeComponentInstance`

Show/Hide Input Window—Showing or Hiding the Floating Input Window (continued)

Data:	A component instance value returned by a prior call to the Component Manager `OpenComponent` function. This value identifies the text service component.
Optional parameter	
Keyword:	`keyAEShowHideInputWindow`
Descriptor type:	`typeBoolean`
Data:	If `TRUE`, the bottomline input window should be shown; if `FALSE`, it should be hidden. This parameter is not needed if the input method is simply inquiring about the state of the input window.
Return parameter	
Keyword:	`keyAEShowHideInputWindow`
Descriptor type:	`typeBoolean`
Data:	The current state of the input window: `TRUE` if the window is shown; `FALSE` if it is hidden. If the optional parameter `keyAEShowHideInputWindow` is included, this return parameter should show the state of the window *before* it was set to the state requested in the optional parameter.

Text Services Manager Routines for Components

This section describes the Text Services Manager component interface—the routines and related data structures that are for the use of text service components. These functions let your text service component

- send Apple events to a client application to request specific information about the active input area in a TSM document
- put up a floating window for various purposes

Sending Apple Events to a Client Application

This section describes the `SendAEFromTSMComponent` function, with which your text service component sends Apple events to a client application.

SendAEFromTSMComponent

The `SendAEFromTSMComponent` function sends Apple events from a text service component to a client application.

```
FUNCTION SendAEFromTSMComponent (VAR theAppleEvent: AppleEvent;
                                 VAR reply: AppleEvent;
                                 sendMode: AESendMode;
```

```
                                    sendPriority: AESendPriority;
                                    timeOutInTicks: LongInt;
                                    idleProc: IdleProcPtr;
                                    filterProc: EventFilterProcPtr):
                                    OSErr;
```

`theAppleEvent`
The Apple event to be sent.

`reply` The reply Apple event returned by `SendAEFromTSMComponent`.

`sendMode` The value that lets you specify one of the following modes specified by corresponding constants: the reply mode for the Apple event, the interaction level, the application switch mode, the reconnection mode, and the return receipt mode. To obtain the value for this parameter, add the appropriate constants. Comprehensive details about these constants are provided in the description of the Apple Event Manager `AESend` function in *Inside Macintosh: Interapplication Communication.*

`sendPriority`
The value that specifies whether to put the Apple event at the back of the event queue (set with the `kAENormalPriority` flag) or at the front of the queue (`kAEHighPriority` flag).

`timeOutInTicks`
The length of time (in ticks) that the client application is willing to wait for the reply or return receipt from the server application before it times out. If the value of this parameter is `kNoTimeOut`, the Apple event never times out.

`idleProc` A pointer to a function for any tasks (such as displaying a globe, a wristwatch, or a spinning beach ball cursor) that the application performs while waiting for a reply or a return receipt.

`filterProc`
A pointer to a routine that accepts certain incoming Apple events that are received while the handler waits for a reply or a return receipt and filters out the rest.

DESCRIPTION

The `SendAEFromTSMComponent` function is essentially a wrapper routine for the Apple Event Manager function `AESend`. See the description of `AESend` for additional necessary information, including constants for the `sendMode` parameter and result codes.

`SendAEFromTSMComponent` identifies your text service component from the `keyAEServerInstance` parameter in the Apple event specified in the `theAppleEvent` parameter. If a reference constant (refcon) in a TSM document that corresponds to this parameter is found in the internal data structures of the Text Services Manager, `SendAEFromTSMComponent` adds the reference constant as the `keyAETSMDocumentRefcon` parameter to the given Apple event before sending it to the application.

If the client application is not TSM-aware, `SendAEFromTSMComponent` routes the Apple events to the floating input window to allow bottomline input.

IMPORTANT

If your text service component changes the environment in any way—such as by modifying the A5 world or changing the current zone—while constructing an Apple event, it must restore the previous settings before sending the Apple event. ▲

Note

Your text service component should always use the `kCurrentProcess` constant as the target address when it creates an Apple event to send to the Text Services Manager. ◆

SEE ALSO

The `AESend` function is described with the Apple Event Manager in *Inside Macintosh: Interapplication Communication.*

The `kCurrentProcess` constant is described in *Inside Macintosh: Processes.*

For sample code showing how a text service component calls the `SendAEFromTSMComponent` function, see Listing 7-11 on page 7-45.

Opening Floating Utility Windows

In conjunction with the Process Manager, the Text Services Manager maintains the floating window service, whose windows occupy a special layer called the floating window service layer. See Figure 7-7 on page 7-14.

The Text Services Manager uses the floating window service to provide a standard floating input window when needed. Text service components can use the service to create, close, and find floating windows used for various other user-interface purposes. You can manipulate the service windows with these calls:

- The `NewServiceWindow` function lets you open a floating window in front of the current application.
- The `CloseServiceWindow` function lets you close a previously allocated floating window.
- The `GetFrontServiceWindow` function helps you find out which is the frontmost window in the floating window service layer.
- The `FindServiceWindow` function helps you find out which part of a text service component's floating window a mouse-down event has occurred in.

Client applications

These calls may be made by client applications also. See the following description of `NewServiceWindow` for special instructions for client applications. ◆

NewServiceWindow

The `NewServiceWindow` function opens a floating utility window in the floating window service layer, in front of the current application. The text service component may use the window for interaction with the user or other purposes.

```
FUNCTION NewServiceWindow (wStorage: Ptr; boundsRect: Rect;
                           title: Str255; visible: Boolean;
                           theProc: Integer; behind: WindowPtr;
                           goAwayFlag: Boolean;
                           ts: ComponentInstance;
                           VAR window: WindowPtr): OSErr;
```

`wStorage` A pointer to the location in memory of the window record. Do not allocate the window record on the stack. Always be sure to allocate the window in the heap, or else pass `NIL` for this parameter.

`boundsRect`
A rectangle given in global coordinates that determines the size and location of the new floating window. This rectangle becomes the `portRect` field of the graphics port record (defined by the QuickDraw `GrafPort` data type) for this window.

`title` A Pascal string that contains the title of the window.

`visible` A Boolean value to determine whether the window is to be drawn. If `TRUE`, `NewServiceWindow` draws the window. First it calls the window definition procedure defined in the `theProc` parameter to draw the window frame. Then it generates an update event for the entire window contents.

`theProc` The window definition procedure for the floating window.

`behind` A window pointer (defined by the Window Manager `WindowPtr` data type) that determines the plane of the floating window. `NewServiceWindow` inserts the new window behind the window pointed to by this parameter. To put the new window behind all other windows, use `behind = NIL`. To place it in front of all other windows, `use behind = POINTER(-1)`.

`goAwayFlag`
A Boolean value that determines whether the go-away region should be drawn in the window. If this parameter is `TRUE` and the window is frontmost (as specified by the `behind` parameter), `NewServiceWindow` draws a go-away region in the frame.

`ts` A component instance returned by a prior call to the Component Manager `OpenComponent` function. This value is stored in the `refcon` field of the window record; text service components should not change the value of the window's `refcon` field.

Client applications

If you are a client application making this call, pass the Process Manager constant `kCurrentProcess` in this parameter so that events in the new window will be forwarded to you. After you have created the window, you can use its `refcon` field for private storage as usual. ◆

`window` A pointer to the newly allocated floating window.

DESCRIPTION

This function calls the Window Manager `NewWindow` function. If a floating window is successfully allocated, `NewServiceWindow` returns a pointer to that window as the function result. Otherwise, it returns `NIL`.

A text service component can open multiple windows in this layer. When a text service component receives an event, it determines whether the event belongs to one of its text service component windows by calling `FindServiceWindow`.

If you are an application that uses `NewServiceWindow` to open a floating window, be sure to hide the floating window when you are switched out; that is, when another application's windows become active.

Balloon Help

If you are writing a text service component and want the service window to have custom Balloon Help, place an `'hwin'` resource (with references to `'hcrt'` and `'STR#'` resources) in your component resource fork, with a name equal to the window title. The Text Services Manager will then open the resources automatically when needed. If you are writing a client application, you need not follow anything other than normal procedures to have Balloon Help. ◆

RESULT CODES

`noErr`	No error
`memFullErr`	Insufficient memory to open the window

SEE ALSO

Window definition procedures and the `NewWindow` function are described in the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.

Balloon Help is described in the Help Manager chapter of *Inside Macintosh: More Macintosh Toolbox*.

CloseServiceWindow

The `CloseServiceWindow` function closes a previously allocated floating input window.

```
FUNCTION CloseServiceWindow (window: WindowPtr): OSErr;
```

`window` A pointer to the service window to close. This function calls the Window Manager `CloseWindow` procedure.

DESCRIPTION

If the window pointer is `NIL` or if it points to a non-floating window, `CloseServiceWindow` returns `paramErr`.

RESULT CODES

`noErr`	No error
`paramErr`	Parameter error

SEE ALSO

The `CloseWindow` procedure is described in the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials.*

GetFrontServiceWindow

The `GetFrontServiceWindow` function determines which is the frontmost window in the floating window service layer.

```
FUNCTION GetFrontServiceWindow (VAR window: WindowPtr): OSErr;
```

`window` A pointer to the frontmost window in the service layer.

DESCRIPTION

This function calls the Window Manager `FrontWindow` function. The `GetFrontServiceWindow` function returns a pointer to the frontmost window in the service layer. If there is no window in the service layer, it returns `NIL`.

SEE ALSO

The `FrontWindow` function is described in the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials.*

FindServiceWindow

The `FindServiceWindow` function determines which part of a text service component's floating window a mouse-down event has occurred in.

```
FUNCTION FindServiceWindow (thePoint: Point;
                            VAR theWindow: WindowPtr): Integer;
```

`thePoint` The point where the mouse button was pressed (in global coordinates, as stored in the `where` field of the Event Manager event record).

`theWindow` A pointer to a Window Manager window pointer (defined by the `WindowPtr` data type) that identifies the floating window in which the mouse-down event occurred. If the mouse-down event did not occur in a text service component floating window, this parameter is set to `NIL`.

DESCRIPTION

The `FindServiceWindow` function is similar to the Window Manager `FindWindow` function, except that `FindServiceWindow` searches the floating window service layer only.

`FindServiceWindow` calls the Window Manager `FindWindow` function. It returns one of the following predefined constants to identify the location of the mouse-down event.

Constant	Value	Explanation
`inDesk`	0	None of the following
`inMenuBar`	1	In menu bar
`inSysWindow`	2	In system window
`inContent`	3	In content region (except grow, if active)
`inDrag`	4	In drag region
`inGrow`	5	In grow region (active window only)
`inGoAway`	6	In go-away region (active window only)
`inZoomIn`	7	In zoom-in region
`inZoomOut`	8	In zoom-out region

It the mouse position is not over a floating window, `FindServiceWindow` returns `inDesk` (0) as its function result, and sets the return parameter `theWindow` to `NIL`.

SEE ALSO

The `FindWindow` function is described in the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials.*

Event records are described in the Event Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials.*

The Process Manager is described in *Inside Macintosh: Processes.*

Text Service Component Routines

This section describes the component-level routines and related data structures and constants through which the Text Services Manager communicates with text service components. The Text Services Manager uses the Component Manager to dispatch the text service component routines to specific text service components.

Client applications also may make the calls described in this section, but the Text Services Manager does not play a role in the connection between the client application making the call and the text service component receiving it. If you are an application making these calls, you need to know the component instance of the component whose routine you are calling.

If you are writing a text service component, it must implement routines for the calls described in this section. With these routines, your component

- provides a text service
- accepts events and updates its cursor and menu (if any)
- confirms active input when requested (if it is an input method)
- identifies the scripts and languages it supports

Providing a Text Service

This section describes the functions a text service component supports to initiate, activate, deactivate, and terminate a text service. The Text Services Manager makes these calls to components either on its own or in response to application-interface calls it receives from client applications.

InitiateTextService

The `InitiateTextService` function instructs a specified text service component to do whatever it needs to set up its operations and commence its performance.

```
FUNCTION InitiateTextService (ts: ComponentInstance):
                                ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

DESCRIPTION

The Text Services Manager can call `InitiateTextService` to any component that it has already opened with the Component Manager `OpenComponent` or `OpenDefaultComponent` functions. Text service components should be prepared to handle `InitiateTextService` calls at any time.

Any text service component can receive multiple `InitiateTextService` calls. The Text Services Manager calls `InitiateTextService` each time the user adds a text service to a TSM document, even if the text service component has already been opened. This provides an opportunity for the component to restart or to display user interface elements that the user may have closed.

This function should return a `ComponentResult` value of zero if there is no error, and an error code if there is one.

ActivateTextService

The `ActivateTextService` function notifies a text service component that its associated document window is becoming active. This allows the text service component to display any associated floating windows.

```
FUNCTION ActivateTextService (ts: ComponentInstance):
                                ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

DESCRIPTION

The appropriate response to `ActivateTextService` is for the text service component to restore its active state, including displaying all floating windows if they have been hidden. If it is an input method, it should specify the redisplay of any unconfirmed text currently in the active input area.

DeactivateTextService

The `DeactivateTextService` function lets a text service component know that its associated document window is becoming inactive. This allows time for the text service component to prepare for deactivation.

```
FUNCTION DeactivateTextService (ts: ComponentInstance):
                                 ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

DESCRIPTION

When it receives a `DeactivateTextService` call, the text service component is responsible for saving whatever state information it needs to save, so that it can restore the proper information when it becomes active again. A component other than an input method should also hide all its floating windows and menus. However, an input-method component should *not* hide its windows in response to this call. If the subsequent document being activated is using the same component's service, it would be irritating to the user to hide and then immediately redisplay the same windows. An input-method component should hide its windows only in response to a `HidePaletteWindows` call.

An input method should not confirm any unconfirmed text in the active input area, but should save it until reactivated.

HidePaletteWindows

The `HidePaletteWindows` function instructs an input method to hide its floating windows because another input method is becoming active.

```
FUNCTION HidePaletteWindows (ts: ComponentInstance):
                              ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

DESCRIPTION

The `HidePaletteWindows` function is not called every time a component's document becomes inactive; it is called by the Text Services Manager only if the new document that is becoming active does not use the same text service component as the document last deactivated. When it receives a `HidePaletteWindows` call, the text service component should hide all its floating and nonfloating windows. Its menus, if any, will be removed from the menu bar by the Text Services Manager.

If the text service component has no palettes, it should return a `ComponentResult` value of `noErr`.

TerminateTextService

The `TerminateTextService` function terminates the operations of a text service in preparation for closing the text service component.

```
FUNCTION TerminateTextService (ts: ComponentInstance):
                               ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

DESCRIPTION

The Text Services Manager calls `TerminateTextService` before closing the component instance. A text service component must use this opportunity to confirm any inline input in progress.

If the text service component needs to remain open, it should return an `OSErr` value in the component result return value. This could happen, for example, if the user chooses Cancel in response to a text service component dialog box.

If this call is made to the *last* open instance of a text service component, the component should hide any open palette windows. If it is an input method, the component should *not* dispose of its menu handle if it has a menu.

Responding to Events and Updating the Cursor and Menu

To pass events to text service components, the Text Services Manager calls the `TextServiceEvent` function. To allow components to handle menu commands, it calls `TextServiceMenuSelect`. To allow components to set the shape of the cursor, it calls `SetTextServiceCursor`. To allow components to add their menus to the menu bar, it calls `GetTextServiceMenu`.

TextServiceEvent

The `TextServiceEvent` function routes an event to a specified text service component.

```
FUNCTION TextServiceEvent (ts: ComponentInstance;
                           numOfEvents: Integer;
                           VAR event: EventRecord):
                           ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

`numOfEvents`
The number of events being passed.

`event` The Event Manager event record (defined by the `EventRecord` data type) for the event being passed.

DESCRIPTION

If the text service component handles the event, it should return a nonzero value for `componentResult` and it should change the event to a null event. If it does not handle the event, it should return 0.

TextServiceMenuSelect

The `TextServiceMenuSelect` function lets a text service component handle commands from its menus.

```
FUNCTION TextServiceMenuSelect (ts: ComponentInstance;
                                serviceMenu: MenuHandle;
                                item: Integer): ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

`serviceMenu`
A Menu Manager menu handle (defined by the `MenuHandle` data type) to a specific text service component menu.

`item` The text service component menu item that the user has selected.

DESCRIPTION

When the user makes a menu selection, the client application calls `TSMMenuSelect`; the Text Services Manager in turn calls `TextServiceMenuSelect` to all active components. The text service component receiving this call should return 0 for `componentResult` if it did not handle the menu selection, and 1 if it did.

After the text service component performs the chosen task, it is not responsible for removing the highlighting from the menu title.

SetTextServiceCursor

The `SetTextServiceCursor` function lets the text service component control the shape of the cursor.

```
FUNCTION SetTextServiceCursor (ts: ComponentInstance;
                               mousePos: Point): ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

`mousePos` A location (specified as a QuickDraw point) that specifies the global coordinates for the vertical and horizontal position of the mouse.

DESCRIPTION

The text service component must return a nonzero value for `ComponentResult` if it has set the cursor, and 0 if it has not.

GetTextServiceMenu

The `GetTextServiceMenu` function returns a handle to a menu belonging to a text service component.

```
FUNCTION GetTextServiceMenu (ts: ComponentInstance;
                             VAR serviceMenu: MenuHandle):
                             ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

`serviceMenu`
A menu handle (defined by the Menu Manager `MenuHandle` data type) for the text service component that is to be updated.

DESCRIPTION

The Text Services Manager calls `GetTextServiceMenu` to a text service component when the component is opened or activated, so that it can put the component's menu on the menu bar.

The menu handle passed in `serviceMenu` may be preallocated or it may be `NIL`. If the menu handle is `NIL`, the text service component should allocate a new menu and return it.

Note
All instances of an input-method component must share a single menu handle, allocated in the system heap. ◆

If the text service component does not have a menu, it should return a `ComponentResult` value of `TSMHasNoMenuErr`.

Confirming Active Input in a TSM Document

To stop active input in a text service component, the Text Services Manager calls the `FixTextService` function described in this section.

FixTextService

The `FixTextService` function explicitly terminates any input that is in progress in a specified text service component.

```
FUNCTION FixTextService (ts: ComponentInstance): ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

DESCRIPTION

This function is equivalent to the user explicitly confirming text, but the request comes instead from the application or from the Text Services Manager. The text service component must stop accepting further input and confirm the current input, as appropriate.

Identifying the Supported Scripts and Languages

The Text Services Manager or a client application may call the `GetScriptLanguageSupport` function to find out all the scripts and languages supported by your text service component.

GetScriptLanguageSupport

The `GetScriptLanguageSupport` function determines which languages and scripts a specified text service component supports, including its primary language and script.

```
FUNCTION GetScriptLanguageSupport (ts: ComponentInstance;
                                   VAR scriptHandle:
                                   ScriptLanguageSupportHandle):
                                   ComponentResult;
```

`ts` A component instance created by a prior call to the Component Manager `OpenComponent` function.

`scriptHandle`
A handle to a script-language support record. The handle must be either `NIL` or a valid handle. If it is `NIL`, the text service component allocates a new handle. If it is already a valid handle, the text service component resizes it as necessary.

DESCRIPTION

The `GetScriptLanguageSupport` function lets a caller find out all scripts and languages that your text service component supports. `GetScriptLanguageSupport` should return a list of scripts and languages in the `scriptHandle` return parameter. The `ComponentResult` return value should contain 0 if the list is correct, or an error value if an error occurred.

The component should list all its supported scripts and languages, starting with the primary script and language as specified in the `componentFlags` field of its component description record. See page 7-15.

The result is returned in a handle to a script-language support record. See "Identifying the Supported Scripts and Languages" on page 7-42 for a description of the script-language support record.

SEE ALSO

For sample code that shows a text service component responding to the `GetScriptLanguageSupport` function, see Listing 7-10 on page 7-43.

Summary of the Text Services Manager

Pascal Summary

Constants

```
CONST

   kTSMVersion = 1;              {Version of Text Services Manager}
   kTextService = 'tsvc';        {Component type for component description}
   kInputMethodService = 'inpm'; {Component subtype for component desc.}

   bTakeActiveEvent = 15;        {Bit set if component takes activate events}
   bScriptMask = $00007F00;                                    {Bits 8 - 14}
   bLanguageMask = $000000FF;                                  {Bits 0 - 7}
   bScriptLanguageMask = ScriptMask + bLanguageMask;           {Bits 0 - 14}

{Hilite styles}
   kCaretPosition = 1;           {specify caret position}
   kRawText = 2;                 {specify range of raw text}
   kSelectedRawText = 3;         {specify range of selected raw text}
   kConvertedText = 4;           {specify range of converted text}
   kSelectedConvertedText = 5;   {specify range of selected converted text}

{Apple Event constants}

   kTextServiceClass = kTextService;   {Event class}
   kUpdateActiveInputArea = 'updt';    {Update active inline area}
   kPos2Offset = 'p2st';               {Convert global coordinates to }
                                       { character position}
   kOffset2Pos = 'st2p';               {Convert character position to }
                                       { global coordinate}
   kShowHideInputWindow = 'shiw';      {show or hide the input window}

   {Event keywords}
   keyAETSMDocumentRefcon = 'refc'; {TSM document refcon}

   keyAEServerInstance = 'srvi'; {Server instance}
   keyAETheData = 'kdat';        {typeText}
```

```
   keyAEScriptTag = 'sclg';      {Script tag}
   keyAEFixLength = 'fixl';
   keyAEHiliteRange = 'hrng';    {Hilite range array}
   keyAEUpdateRange = 'udng';    {Update range array}
   keyAEClauseOffsets = 'clau';  {Clause offsets array}
   keyAECurrentPoint = 'cpos';   {Current point}
   keyAEDragging = 'bool';       {Dragging flag}
   keyAEOffset = 'ofst';         {Offset}
   keyAERegionClass = 'rgnc';    {Region class}
   keyAEPoint = 'gpos';          {Current point}
   keyAEBufferSize = 'buff';     {Buffer size to get the text}
   keyAERequestedType = 'rtyp';  {Requested text type}
   keyAEMoveView = 'mvvw';       {Move view flag}
   keyAELength = 'leng';         {Length}
   keyAENextBody = 'nxbd';       {Next or previous body}

{optional keywords for Offset2Pos}
   keyAETextFont = 'ktxf';
   keyAETextPointSize = 'ktps';
   keyAETextLineHeight = 'ktlh';
   keyAETextLineAscent = 'ktas';
   keyAEAngle = 'kang';

{optional keyword for Pos2Offset}
   keyAELeftSide = 'klef';       {type Boolean}

{optional keyword for kShowHideInputWindow}
   keyAEShowHideInputWindow = 'shiw';  {type Boolean}

{keyword for PinRange}
   keyAEPinRange = 'pnrg';

{Desc type ...}
   typeComponentInstance = 'cmpi';  {component instance}
   typeTextRange = 'txrn';          {text range}
   typeTextRangeArray = 'tray';     {text range array}
   typeOffsetArray = 'ofay';        {offset array}
   typeIntlWritingCode = 'intl';    {script code}
   typeQDPoint = 'QDpt';            {QuickDraw point}
   typeAEText = 'tTXT';             {Apple event text}
   typeText = 'TEXT';               {plain text}

{Apple event descriptor type constants}
   kTSMOutsideOfBody = 1;
   kTSMInsideOfBody = 2;
```

```
   kTSMInsideOfActiveInputArea = 3;

   kNextBody = 1;
   kPreviousBody = 2;

{Apple event error constants}
   errOffsetInvalid = -1800;
   errOffsetIsOutsideOfView = -1801;
   errTopOfDocument = -1810;
   errTopOfBody = -1811;
   errEndOfDocument = -1812;
   errEndOfBody = -1813;
```

Data Types

```
TYPE TextRange =
   RECORD
      fStart:        LongInt;
      fEnd:          LongInt;
      fHiliteStyle:  Integer;
   END;
   TextRangePtr = ^TextRange;
   TextRangeHandle = ^TextRangePtr;

   TextRangeArray =
   RECORD
      fNumOfRanges:  Integer;
      fRange:        ARRAY [0..0] of TextRange;
   END;
   TextRangeArrayPtr = ^TextRangeArray;
   TextRangeArrayHandle = ^TextRangeArrayPtr;

   OffsetArray =
   RECORD
      fNumOfOffsets: Integer;
      fOffset:       ARRAY [0..0] of LongInt;
   END;
   OffsetArrayPtr = ^OffsetArray;
   OffsetArrayHandle = ^OffsetArrayPtr;

   TextServiceInfo =
   RECORD
      fComponent: Component;
```

```
      fItemName:  Str255;
   END;
   TextServicesInfoPtr = ^TextServiceInfo;

   TextServiceList =
   RECORD
      fTextServiceCount:   Integer;
      fServices:           ARRAY [0..0] of TextServiceInfo;
   END;
   TextServiceListPtr = ^TextServiceList;
   TextServiceListHandle = ^TextServiceListPtr;

   ScriptLanguageRecord =
   RECORD
      fScript:    ScriptCode;
      fLanguage:  LangCode;
   END;

   ScriptLanguageSupport =
   RECORD
      fScriptLanguageCount:   Integer;
      fScriptLanguageArray:   ARRAY [0..0] of ScriptLanguageRecord;
   END;
   ScriptLanguageSupportPtr = ^ScriptLanguageSupport;
   ScriptLanguageSupportHandle = ^ScriptLanguageSupportPtr;

   InterfaceTypeList = ARRAY [0..0] of OSType;

   TSMDocumentID = Ptr;
```

Text Services Manager Routines for Client Applications

Initializing and Closing as a TSM-Aware Application

```
FUNCTION InitTSMAwareApplication: OSErr;
FUNCTION CloseTSMAwareApplication: OSErr;
```

Creating and Activating TSM Documents

```
FUNCTION NewTSMDocument     (numOfInterface: Integer;
                             VAR supportedInterfaceTypes: InterfaceTypeList;
                             VAR idocID: TSMDocumentID;
                             refCon: LongInt): OSErr;
FUNCTION ActivateTSMDocument
                            (idocID: TSMDocumentID): OSErr;
```

```
FUNCTION DeactivateTSMDocument
                            (idocID: TSMDocumentID): OSErr;
FUNCTION DeleteTSMDocument  (idocID: TSMDocumentID): OSErr;
```

Passing Events to Text Service Components

```
FUNCTION TSMEvent           (VAR event: EventRecord): Boolean;
```

Passing Menu Selections and Cursor Setting

```
FUNCTION TSMMenuSelect      (menuResult: LongInt): Boolean;
FUNCTION SetTSMCursor       (mousePos: Point): Boolean;
```

Confirming Active Input in a TSM Document

```
FUNCTION FixTSMDocument     (idocID: TSMDocumentID): OSErr;
```

Making Text Services Available to the User

```
FUNCTION GetServiceList     (numOfInterfaceTypes: Integer;
                             supportedInterfaceTypes: InterfaceTypeList;
                             VAR serviceInfo: TextServiceListHandle;
                             VAR seedValue: LongInt): OSErr;
FUNCTION OpenTextService    (idocID: TSMDocumentID; aComponent: Component;
                             VAR aComponentInstance: ComponentInstance):
                             OSErr;
FUNCTION CloseTextService   (idocID: TSMDocumentID; aComponentInstance:
                             ComponentInstance): OSErr;
```

Requesting a Floating Input Window

```
FUNCTION UseInputWindow     (idocID: TSMDocumentID; useWindow: Boolean):
                             OSErr;
```

Associating Scripts and Languages With Components

```
FUNCTION SetDefaultInputMethod
                            (ts: Component;
                             VAR slRecord: ScriptLanguageRecord): OSErr;
FUNCTION GetDefaultInputMethod
                            (VAR ts: Component;
                             VAR slRecord: ScriptLanguageRecord): OSErr;
FUNCTION SetTextServiceLanguage
                            (VAR slRecord: ScriptLanguageRecord): OSErr;
FUNCTION GetTextServiceLanguage
                            (VAR slRecord: ScriptLanguageRecord): OSErr;
```

Text Services Manager Routines for Components

Sending Apple Events to a Client Application

```
FUNCTION SendAEFromTSMComponent
                            (VAR theAppleEvent: AppleEvent;
                             VAR reply: AppleEvent; sendMode: AESendMode;
                             sendPriority: AESendPriority;
                             timeOutInTicks: LongInt;
                             idleProc: IdleProcPtr;
                             filterProc: EventFilterProcPtr): OSErr;
```

Opening Floating Utility Windows

```
FUNCTION NewServiceWindow   (wStorage: Ptr; boundsRect: Rect;
                             title: Str255; visible: Boolean;
                             theProc: Integer; behind: WindowPtr;
                             goAwayFlag: Boolean; ts: ComponentInstance;
                             VAR window: WindowPtr): OSErr;
FUNCTION CloseServiceWindow (window: WindowPtr): OSErr;
FUNCTION GetFrontServiceWindow
                            (VAR window: WindowPtr): OSErr;
FUNCTION FindServiceWindow  (thePoint: Point; VAR theWindow: WindowPtr):
                             Integer;
```

Text Service Component Routines

Providing a Text Service

```
FUNCTION InitiateTextService
                            (ts: ComponentInstance): ComponentResult;
FUNCTION ActivateTextService
                            (ts: ComponentInstance): ComponentResult;
FUNCTION DeactivateTextService
                            (ts: ComponentInstance): ComponentResult;
FUNCTION HidePaletteWindows (ts: ComponentInstance): ComponentResult;
FUNCTION TerminateTextService
                            (ts: ComponentInstance): ComponentResult;
```

Responding to Events and Updating the Cursor and Menu

```
FUNCTION TextServiceEvent   (ts: ComponentInstance; numOfEvents: Integer;
                             VAR event: EventRecord): ComponentResult;
```

```
FUNCTION TextServiceMenuSelect
                            (ts: ComponentInstance; serviceMenu:
                             MenuHandle; item: Integer): ComponentResult;
FUNCTION SetTextServiceCursor
                            (ts: ComponentInstance; mousePos: Point):
                             ComponentResult;
FUNCTION GetTextServiceMenu (ts: ComponentInstance;
                             VAR serviceMenu: MenuHandle): ComponentResult;
```

Confirming Active Input in a TSM Document

```
FUNCTION FixTextService     (ts: ComponentInstance): ComponentResult;
```

Identifying the Supported Scripts and Languages

```
FUNCTION GetScriptLanguageSupport
                            (ts: ComponentInstance; VAR scriptHandle:
                             ScriptLanguageSupportHandle): ComponentResult;
```

C Summary

Constants

```
#define  kTSMVersion          1              /* Version of the
                                               Text Services Manager */
#define  kTextService         'tsvc'         /* component type for
                                               the component description */
#define  kInputMethodService  'inpm'         /* component subtype for
                                               the component description */

#define  bTakeActiveEvent     15             /* bit set if the component
                                                takes active event */
#define  bScriptMask          0x00007F00     /* bit 8 - 14 */
#define  bLanguageMask        0x000000FF     /* bit 0 - 7  */
#define  bScriptLanguageMask  bScriptMask+bLanguageMask     /* bit 0 - 14  */

/* Hilite styles ... */
typedef enum {
   kCaretPosition          =  1,    /* specify caret position */
   kRawText                =  2,    /* specify range of raw text */
   kSelectedRawText        =  3,    /* specify range of selected raw text */
   kConvertedText          =  4,    /* specify range of converted text */
```

```
   kSelectedConvertedText  =  5     /* specify range of selected
                                       converted text */
} HiliteStyleType;

/* Apple Event constants ... */

/* Event class ... */
#define     kTextServiceClass       kTextService

/* event ID ... */
#define     kUpdateActiveInputArea  'updt'   /* update active Inline area */
#define     kPos2Offset             'p2st'   /* converting global coordinates
                                                to char position */
#define     kOffset2Pos             'st2p'   /* converting char position
                                                to global coordinates */
#define     kShowHideInputWindow    'shiw'   /* show or hide bottomline
                                                input window */

/* Event keywords ... */
#define  keyTSMDocumentRefcon    'refc'      /* TSM document refcon */
#define  keyAEServerInstance     'srvi'      /* component instance */
#define  keyAETheData            'kdat'      /* typeText */
#define  keyAEScriptTag          'sclg'      /* script tag */
#define  keyAEFixLength          'fixl'      /* fix len ?? */
#define  keyAEHiliteRange        'hrng'      /* hilite range array */
#define  keyAEUpdateRange        'udng'      /* update range array */
#define  keyAEClauseOffsets      'clau'      /* Clause Offsets array */
#define  keyAECurrentPoint       'cpos'      /* current point */
#define  keyAEDragging           'bool'      /* dragging flag */
#define  keyAEOffset             'ofst'      /* offset */
#define  keyAERegionClass        'rgnc'      /* region class */
#define  keyAEPoint              'gpos'      /* current point */
#define  keyAEBufferSize         'buff'      /* buffer size to get text */
#define  keyAERequestedType      'rtyp'      /* requested text type */
#define  keyAEMoveView           'mvvw'      /* move view flag */
#define  keyAELength             'leng'      /* length */
#define  keyAENextBody           'nxbd'      /* next or previous body */

/* optional keyword for UpdateActiveInputArea */
#define  keyAEPinRange           'pnrg'

/* optional keywords for Offset2Pos */
#define  keyAETextFont           'ktxf'
#define  keyAETextPointSize      'ktps'
```

```
#define  keyAETextLineHeight     'ktlh'
#define  keyAETextLineAscent     'ktas'
#define  keyAEAngle              'kang'

/* optional keywords for Pos2Offset */
#define  keyAELeadingEdge        'klef'

/* Apple event descriptor type ... */
#define  typeComponentInstance   'cmpi'      /* server instance */
#define  typeTextRange           'txrn'      /* text range record */
#define  typeTextRangeArray      'tray'      /* text range array */
#define  typeOffsetArray         'ofay'      /* offset array */
#define  typeIntlWritingCode     'intl'      /* script code */
#define  typeQDPoint             'QDpt'      /* QuickDraw Point */
#define  typeAEText              'tTXT'      /* Apple Event text */
#define  typeText                'TEXT'      /* Plain text */
#define  typeFixed               'fixd'      /* Fixed number 16.16 */

/* Apple event descriptor type constants */
typedef enum {
   kTSMOutsideOfBody             = 1,
   kTSMInsideOfBody              = 2,
   kTSMInsideOfActiveInputArea   = 3
} AERegionClassType;

typedef enum {
   kNextBody                     = 1,
   kPreviousBody                 = 2
} AENextBodyType;

/* Apple Event error definitions */
typedef enum {
   errOffsetInvalid           = -1800,
   errOffsetIsOutsideOfView   = -1801,
   errTopOfDocument           = -1810,
   errTopOfBody               = -1811,
   errEndOfDocument           = -1812,
   errEndOfBody               = -1813
} AppleEventErrorType;
```

Data Types

```
struct TextRange {                  /* typeTextRange  'txrn' */
   long  fStart;
   long  fEnd;
   short fHiliteStyle;
};
typedef struct TextRange      TextRange;
typedef        TextRange      *TextRangePtr;
typedef        TextRangePtr   *TextRangeHandle;

struct TextRangeArray {          /* typeTextRangeArray   'txra' */
   short       fNumOfRanges;     /* specify the size of the fRange array */
   TextRange   fRange[1];        /* when fNumOfRanges > 1, the size of this
                                    array has to be calculated */
};
typedef struct TextRangeArray    TextRangeArray;
typedef        TextRangeArray    *TextRangeArrayPtr;
typedef        TextRangeArrayPtr *TextRangeArrayHandle;

struct OffsetArray {             /* typeOffsetArray'offa' */
   short fNumOfOffsets;          /* specify the size of the fOffset array */
   long  fOffset[1];             /* when fNumOfOffsets > 1, the size of
                                    this array has to be calculated */
};
typedef struct OffsetArray    OffsetArray;
typedef        OffsetArray    *OffsetArrayPtr;
typedef        OffsetArrayPtr *OffsetArrayHandle;

/* extract Script/Language code from Component flag ... */
#define  mGetScriptCode(cdRec)   ((ScriptCode)  (cdRec.componentFlags &
                                                      bScriptMask) >> 8)
#define  mGetLanguageCode(cdRec) ((LangCode)    cdRec.componentFlags &
                                                         bLanguageMask)

typedef  void  *TSMDocumentID;

/* text service component information list */
struct TextServiceInfo {
   Component   fComponent;
   Str255      fItemName;
};
typedef struct TextServiceInfo   TextServiceInfo;
typedef        TextServiceInfo   *TextServiceInfoPtr;
```

```
/*text service component list*/
struct TextServiceList {
   short             fTextServiceCount;   /* number of entries in the
                                             'fServices' array */
   TextServiceInfo   fServices[1];        /* Note: array of 'TextServiceInfo'
                                             records follows */
};
typedef struct TextServiceList      TextServiceList;
typedef        TextServiceList      *TextServiceListPtr;
typedef        TextServiceListPtr   *TextServiceListHandle;

/*script and language record*/
struct ScriptLanguageRecord {
   ScriptCode  fScript;
   LangCode    fLanguage;
};
typedef struct ScriptLanguageRecord ScriptLanguageRecord;

/*script and language support record*/
struct ScriptLanguageSupport {
 short                  fScriptLanguageCount; /* number of entries in the
                                                 'fScriptLanguageArray'
                                                 array */
 ScriptLanguageRecord   fScriptLanguageArray[1]; /* Note: array of
                                                    'ScriptLanguageRecord'
                                                    records follows */
};
typedef struct ScriptLanguageSupport      ScriptLanguageSupport;
typedef        ScriptLanguageSupport      *ScriptLanguageSupportPtr;
typedef        ScriptLanguageSupportPtr   *ScriptLanguageSupportHandle;
```

Text Services Manager Routines for Client Applications

Initializing and Closing as a TSM-Aware Application

```
pascal OSErr InitTSMAwareApplication ();
pascal OSErr CloseTSMAwareApplication ();
```

Creating and Activating TSM Documents

```
pascal OSErr NewTSMDocument (short numOfInterface,
                             OSType supportedInterfaceTypes[],
                             TSMDocumentID *idocID, long refCon);
```

```
pascal OSErr ActivateTSMDocument
                            (TSMDocumentID idocID);
pascal OSErr DeactivateTSMDocument
                            (TSMDocumentID idocID);
pascal OSErr DeleteTSMDocument
                            (TSMDocumentID idocID);
```

Passing Events to Text Service Components

```
pascal Boolean TSMEvent     (EventRecord *event);
```

Passing Menu Selections and Cursor Setting

```
pascal Boolean TSMMenuSelect(long menuResult);
pascal Boolean SetTSMCursor (Point mousePos);
```

Confirming Active Input in a TSM Document

```
pascal OSErr FixTSMDocument (TSMDocumentID idocID);
```

Making Text Services Available to the User

```
pascal OSErr GetServiceList (short numOfInterfaceTypes,
                             OSType supportedInterfaceTypes[],
                             TextServiceListHandle *serviceInfo,
                             long *seedValue);
pascal OSErr OpenTextService
                            (TSMDocumentID idocID, Component aComponent,
                              ComponentInstance *aComponentInstance);
pascal OSErr CloseTextService
                            (TSMDocumentID idocID,
                             ComponentInstance aComponentInstance)
```

Requesting a Floating Input Window

```
pascal OSErr UseInputWindow (TSMDocumentID idocID, Boolean useWindow);
```

Associating Scripts and Languages With Components

```
pascal OSErr SetDefaultInputMethod
                            (Component  ts,
                             ScriptLanguageRecord *slRecordPtr);
pascal OSErr GetDefaultInputMethod
                            (Component *ts,
                             ScriptLanguageRecord *slRecordPtr);
```

```
pascal OSErr SetTextServiceLanguage
                            (ScriptLanguageRecord *slRecordPtr);
pascal OSErr GetTextServiceLanguage
                            (ScriptLanguageRecord *slRecordPtr);
```

Text Services Manager Routines for Components

Sending Apple Events to a Client Application

```
pascal OSErr SendAEFromTSMComponent
                            (AppleEvent *theAppleEvent,
                             AppleEvent *reply, AESendMode sendMode,
                             AESendPriority sendPriority,
                             long timeOutInTicks, IdleProcPtr idleProc,
                             EventFilterProcPtr filterProc);
```

Opening Floating Utility Windows

```
pascal OSErr NewServiceWindow
                            (void *wStorage, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short theProc, WindowPtr behind,
                             Boolean goAwayFlag, ComponentInstance ts,
                             WindowPtr *window);
pascal OSErr CloseServiceWindow
                            (WindowPtr window);
pascal OSErr GetFrontServiceWindow
                            (WindowPtr *window);
pascal short FindServiceWindow
                            (Point thePoint, WindowPtr *theWindow);
```

Text Service Component Routines

Providing a Text Service

```
pascal ComponentResult InitiateTextService
                            (ComponentInstance ts);
pascal ComponentResult ActivateTextService
                            (ComponentInstance ts);
pascal ComponentResult DeactivateTextService
                            (ComponentInstance ts);
```

```
pascal ComponentResult HidePaletteWindows
                            (ComponentInstance ts);
pascal ComponentResult TerminateTextService
                            (ComponentInstance ts);
```

Responding to Events and Updating the Cursor and Menu

```
pascal ComponentResult TextServiceEvent
                            (ComponentInstance ts,
                             short numOfEvents, EventRecord *event)
pascal ComponentResult TextServiceMenuSelect
                            (ComponentInstance ts,
                             MenuHandle serviceMenu, short item);
pascal ComponentResult SetTextServiceCursor
                            (ComponentInstance ts, Point mousePos);
pascal ComponentResult GetTextServiceMenu
                            (ComponentInstance ts, MenuHandle *serviceMenu);
```

Confirming Active Input in a TSM Document

```
pascal ComponentResult FixTextService
                            (ComponentInstance ts);
```

Identifying the Supported Scripts and Languages

```
pascal ComponentResult GetScriptLanguageSupport
                            (ComponentInstance ts,
                             ScriptLanguageSupportHandle *scriptHdl);
```

Assembly-Language Summary

Trap Macros

Trap Macro Names for Text Services Manager Routines

Pascal name	Trap macro name
NewTSMDocument	_NewTSMDocument
DeleteTSMDocument	_DeleteTSMDocument
ActivateTSMDocument	_ActivateTSMDocument
DeactivateTSMDocument	_DeactivateTSMDocument
TSMEvent	_TSMEvent
TSMMenuSelect	_TSMMenuSelect

Pascal name	Trap macro name
SetTSMCursor	_SetTSMCursor
FixTSMDocument	_FixTSMDocument
GetServiceList	_GetServiceList
OpenTextService	_OpenTextService
CloseTextService	_CloseTextService
SendAEFromTSMComponent	_SendAEFromTSMComponent
SetDefaultInputMethod	_SetDefaultInputMethod
GetDefaultInputMethod	_GetDefaultInputMethod
SetTextServiceLanguage	_SetTextServiceLanguage
GetTextServiceLanguage	_GetTextServiceLanguage
UseInputWindow	_UseInputWindow
NewServiceWindow	_NewServiceWindow
CloseServiceWindow	_CloseServiceWindow
GetFrontServiceWindow	_GetFrontServiceWindow
InitTSMAwareApplication	_InitTSMAwareApplication
CloseTSMAwareApplication	_CloseTSMAwareApplication
FindServiceWindow	_FindServiceWindow

Trap Macro Names for Text Service Component Routines

Pascal name	Trap macro name
GetScriptLanguageSupport	_GetScriptLanguageSupport
InitiateTextService	_InitiateTextService
TerminateTextService	_TerminateTextService
ActivateTextService	_ActivateTextService
DeactivateTextService	_DeactivateTextService
TextServiceEvent	_TextServiceEvent
GetTextServiceMenu	_GetTextServiceMenu
TextServiceMenuSelect	_TextServiceMenuSelect
FixTextService	_FixTextService
SetTextServiceCursor	_SetTextServiceCursor
HidePaletteWindows	_HidePaletteWindows

Result Codes

`tsmComponentNoErr`	0	Component result: no error
`tsmUnsupScriptLanguageErr`	–2500	Specified script and language are not supported
`tsmInputMethodNotFoundErr`	–2501	Specified input method cannot be found
`tsmNotAnAppErr`	–2502	The caller was not an application
`tsmAlreadyRegisteredErr`	–2503	The caller is already TSM-initialized
`tsmNeverRegisteredErr`	–2504	The caller is not TSM-aware
`tsmInvalidDocIDErr`	–2505	Invalid TSM document ID
`tsmTSMDocBusyErr`	–2506	Document is still active
`tsmDocNotActiveErr`	–2507	Document is not active
`tsmNoOpenTSErr`	–2508	There is no open text service component
`tsmCantOpenComponentErr`	–2509	Can’t open the component
`tsmTextServiceNotFoundErr`	–2510	No text service component found
`tsmDocumentOpenErr`	–2511	There are open documents
`tsmUseInputWindowErr`	–2512	An input window is being used
`tsmTSHasNoMenuErr`	–2513	The text service component has no menu
`tsmTSNotOpenErr`	–2514	Text service component is not open
`tsmComponentAlreadyOpenErr`	–2515	Text service component already open for document
`tsmInputMethodIsOldErr`	–2516	The default input method is old-style
`tsmScriptHasNoIMErr`	–2517	Script has no (or old) input method
`tsmUnsupportedTypeErr`	–2518	Unsupported interface type
`tsmUnknownErr`	–2519	Any other error