Setting Up Your Carbon Application to Use the Services Menu

Carbon > Interapplication Communication

2003-12-10

Contents

Figures, Tables, and Listings

Introduction to Setting Up Your Carbon Application to Use the Services Menu

Application services give applications an open-ended way to extend each other's functionality by allowing them to:

- Provide services to other applications
- Access functionality provided by other applications

An application that provides a service advertises the operation it can perform on a particular type of data—for example, encryption of text, optical character recognition of a bitmapped image, or generating text such as a message of the day. Any application that signs up to use services automatically accesses the advertised functionality through its Services menu. An application doesn't need to know in advance what operations are available; it merely needs to indicate the types of data it uses, and the Services menu makes available the operations that apply to those types of data.

Who Should Read This Document?

This document describes how application services work, shows some typical Services menus, and provides instructions on how you can use services in your application. You should read this document if you are an application developer and want to provide your application's services to other applications or make services from other applications available to your application.

Before you read this document, you should be familiar with information property lists. You need to know what they are and how to add properties to a list. Carbon developers should also know how to write and install Carbon event handlers.

Organization of This Document

This document is organized as follows:

- "Application Services Concepts" (page 9), discusses the types of services, the Services menu, services properties, and what happens when a service is invoked.
- "Application Services Tasks" (page 17), provides instructions on how to set up your application to use services provided by other applications and how to provide your services to other applications.
- "Carbon Events for Services" (page 35), describes the Carbon event classes, kinds, and event parameters defined for services events.

See Also

- *Apple Human Interface Guidelines* in User Experience Documentation provides guidelines on naming menu items and designing the interface for a services application.
- Handling Carbon Events in Carbon Events & Other Input Documentation describes how to implement Carbon events in your application.
- System Overview in Mac OS X Documentation contains information on property lists and how they are used in Mac OS X.
- *Learning Carbon*, available through O'Reilly and Associates, contains information on writing and using Carbon event handlers as well as information on how to set up and use property lists.

Application Services Concepts

This chapter provides an overview of the Services menu and describes how application services are provided and accessed. It also shows typical examples of the services available to, or provided by, applications in Mac OS X.

Types of Application Services

Mac OS X offers two types of services:

- Processor. This type of service acts on data. A processor service acts on the current selection and then sends it to the service. For example, if a user selects an email address in a TextEdit document, and then chooses Mail > Mail To from the Services menu, Mail copies the person's address, launches the Mail application, and pastes the address into the To field of a new email message.
- Provider. This type of service gives data to the calling application. For example, if a user chooses Grab > Screen from the Services menu, the Grab application opens, takes a screenshot, then returns the screenshot (TIFF data in this case) to the calling application. The calling application (such as TextEdit) is responsible for pasting the data into the active document.

Let's take a look at a few examples. Figure 2-1 (page 9) shows the Services menu from the TextEdit application. Make Sticky is an example of a processor service. The Make Sticky command takes the current selection in the TextEdit document, opens a new Stickies document, and then pastes the selection into the Stickies document.

Figure 1-1 Make Sticky is a processor service

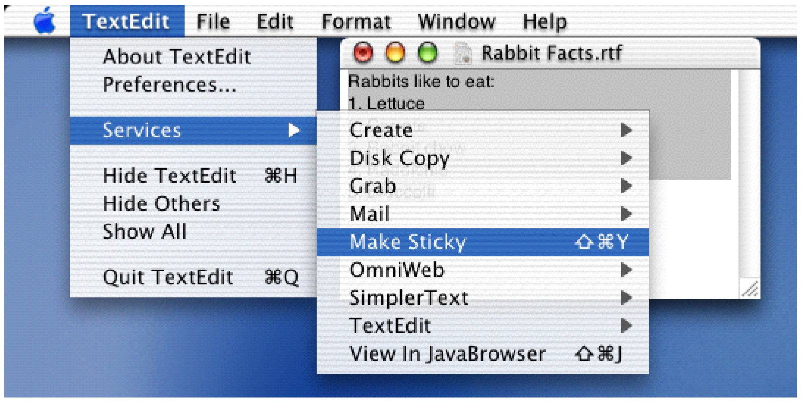
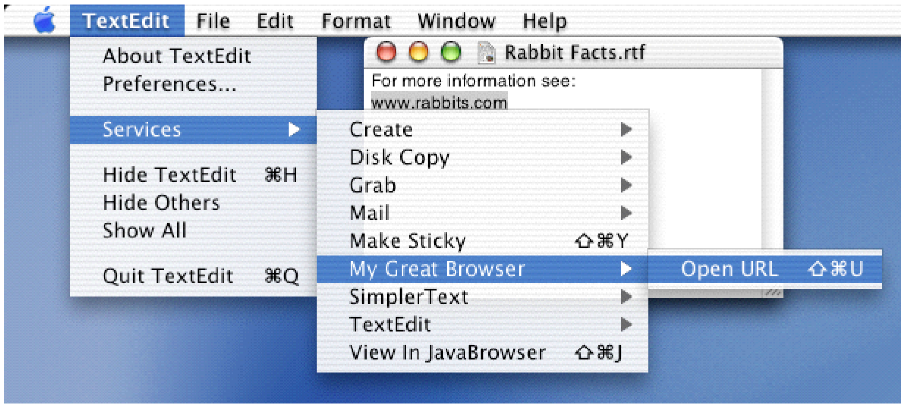


Figure 2-2 (page 10) shows another example of a processor service. In this case, the Open URL command copies the selected text, launches a Web browser, pastes the selected text into the browser's location field, and then tries to connect to that location.

Figure 1-2 Open URL is a processor service



Grab is a provider service. Figure 2-3 (page 10) shows the Wolf Facts document before Grab > Screen is invoked. Figure 2-4 (page 11) shows the Wolf Facts document after Grab has taken a shot of the current screen and returned the data to the TextEdit application. Recall that it is TextEdit's responsibility to do something with the returned data. In this example, TextEdit simply pastes the TIFF into the current document at the active insertion point.

Figure 1-3 Grab is a provider service

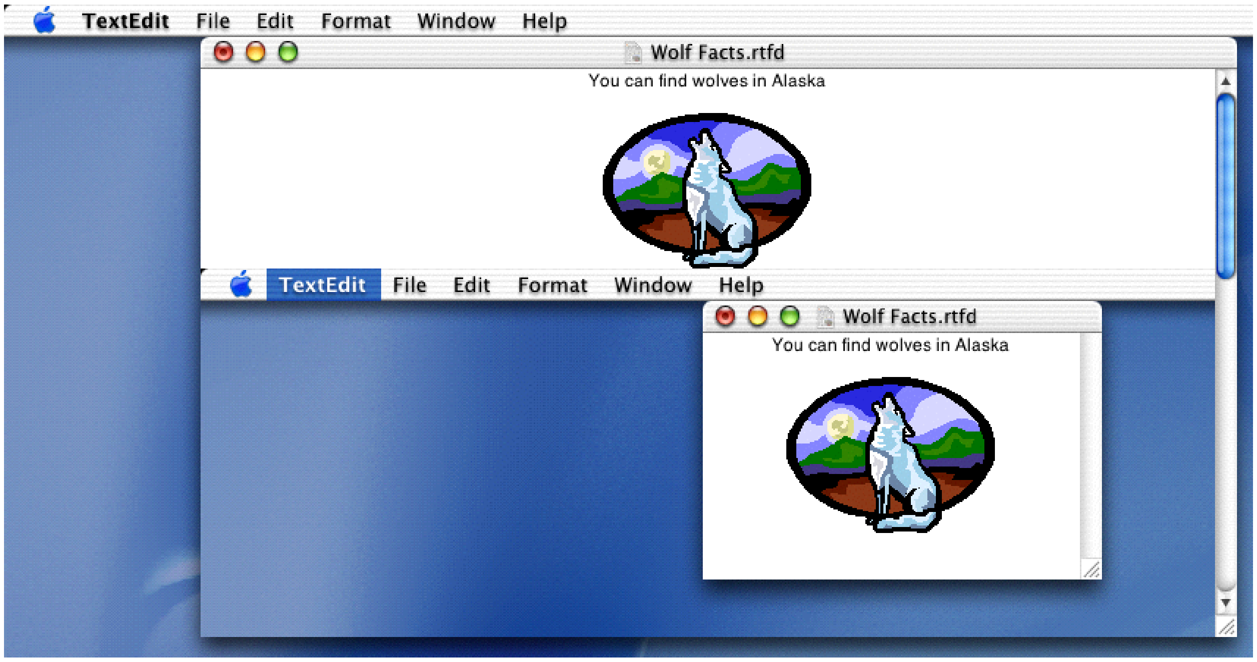
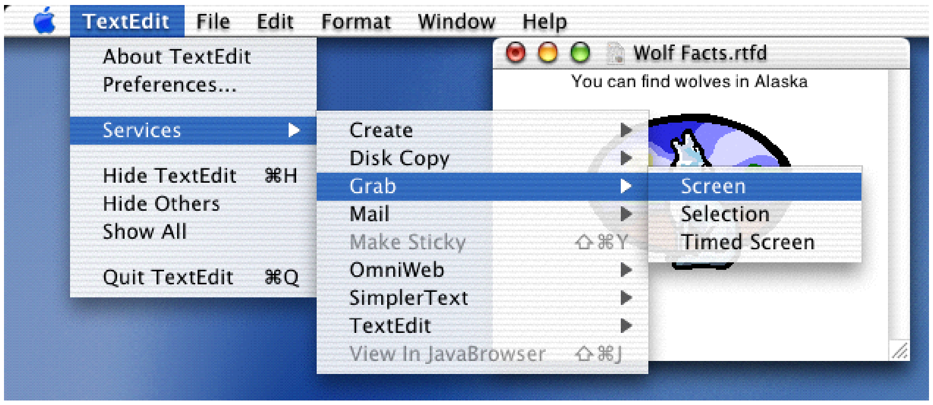


Figure 1-4 The Wolf Facts document after a screenshot has been inserted



Installation Paths for Services

A service can be offered as part of an application, such as Mail, or as a standalone service—one without a user interface that is intended for use only in the Services menu. Applications that offer services should be built with the `.app` extension and installed in the `/Applications` folder. A standalone service should be built with the `.service` extension and stored in the `/Library/Services` folder.

When a user logs in, Mac OS X searches the `/Applications` and `/Library/Services` folders in the four file-system domains—System, Network, Local, and User. (See *Inside Mac OS X: System Overview* for details on file-system domains.) The system examines the information property list for each bundle in these locations and assembles a list of available services (see "Services Properties" (page 12)). It uses this information to populate the items in the Services menu.

Items in the Services Menu

The Services menu appears automatically as an item in the application menu for Carbon and Cocoa applications. (This is a new feature for Carbon applications, starting with Mac OS X version 10.1.) If an application enables services (see "Using a Service" (page 17)), the appropriate items are available in the Services submenu. Otherwise, items in the Services menu are dimmed.

The items in the Services submenu can be commands or submenus that contain commands. The exact wording of the commands and whether or not there is a submenu is specified by the application. Typically, if an application offers only one service, just the service (stated as a command) is listed in the Services menu. For example, the Stickies application offers only one service—making a new Sticky note—so only the command Make Sticky is listed in the Services menu.

If an application offers more than one service, the application's name appears in the Services menu, and the services offered by the application appear in a submenu. For example, the Grab application offers three services: taking a screenshot of the entire screen, taking a screenshot of a selected part of the screen, and taking a screenshot of the entire screen after a set amount of time. As you can see in Figure 2-3 (page 10), Grab is an item in the Services menu that has its own submenu listing the commands that invoke Grab's three services: Screen, Selection, and Timed Screen.

The Services menu for an application is populated with items when the application starts up, but items in the menu aren't enabled until the user chooses Services from the application menu. Choosing Services causes an event that asks the application to supply the data types it handles. For example, if an application only creates or reads plain text files, it should respond to the event by supplying text as the data type the application can handle. Only those services that provide or act on text will be enabled in the Services menu. The other menu items will be dimmed, unavailable for the user to choose.

Services Properties

Any application that has one or more services to provide must advertise the type of data its services can handle. Services are advertised through the `NSServices` property of the application's information property list (`Info.plist` file).

Note: The information property list (`Info.plist`) contains key-value pairs that specify the application's properties that are of interest to the Finder and other applications. Although the `Info.plist` is a text file that uses XML (Extensible Markup Language) format, you should not modify the XML directly unless you are very familiar with XML syntax. Instead, use Project Builder or the Property List Editor application provided with Mac OS X to modify the `Info.plist` file. You can find more information on property lists in *Inside Mac OS X: System Overview*.

`NSServices` is a property whose value is an array of dictionaries that specifies the services provided by the application. Keys for each dictionary entry, are as follows:

- `NSMessage` indicates the name of the service to invoke.
- `NSPortName` is the name of the port to which the application should listen for service requests. Its value depends on how the service provider application is registered. In most cases, this is the application name.
- `NSMenuItem` specifies the text of the Services menu item. You can use a slash to specify a submenu. For example, `Mail/Send Selection` appears in the Services menu as a submenu named Mail with an item named Send Selection. `NSMenuItem` must be unique, as only one is used in the Services menu if there are duplicates.
- `NSKeyEquivalent` is an optional item that specifies the keyboard equivalent that invokes the menu command.
- `NSSendTypes` is an array that contains data type names. Send types are the types sent from the application requesting a service. An applicaton that provides a service must specify an `NSSendType`, an `NSReturnType`, or both.
- `NSReturnTypes` is an array that contains data type names. Return types are the data types returned to the application requesting a service. An applicaton that provides a service must specify an `NSSendType`, an `NSReturnType`, or both.
- `NSUserData` is an optional string that contains a value of your choice. You can use this string to customize the behavior of your service. This entry is useful for applications that provide open-ended services.
- `NSTimeout` is an optional string that indicates the number of milliseconds Services should wait for a response from the application providing a service when a response is required. If wait time exceeds the timeout value, Services displays an error message to the user. If you don't specify this entry, the timeout value is 30000 milliseconds (30 seconds).

Let's take a look at the information property list for the Grab application. The `NSServices` property is shown in Figure 2-5 (page 13) as it appears in the Property List Editor application.

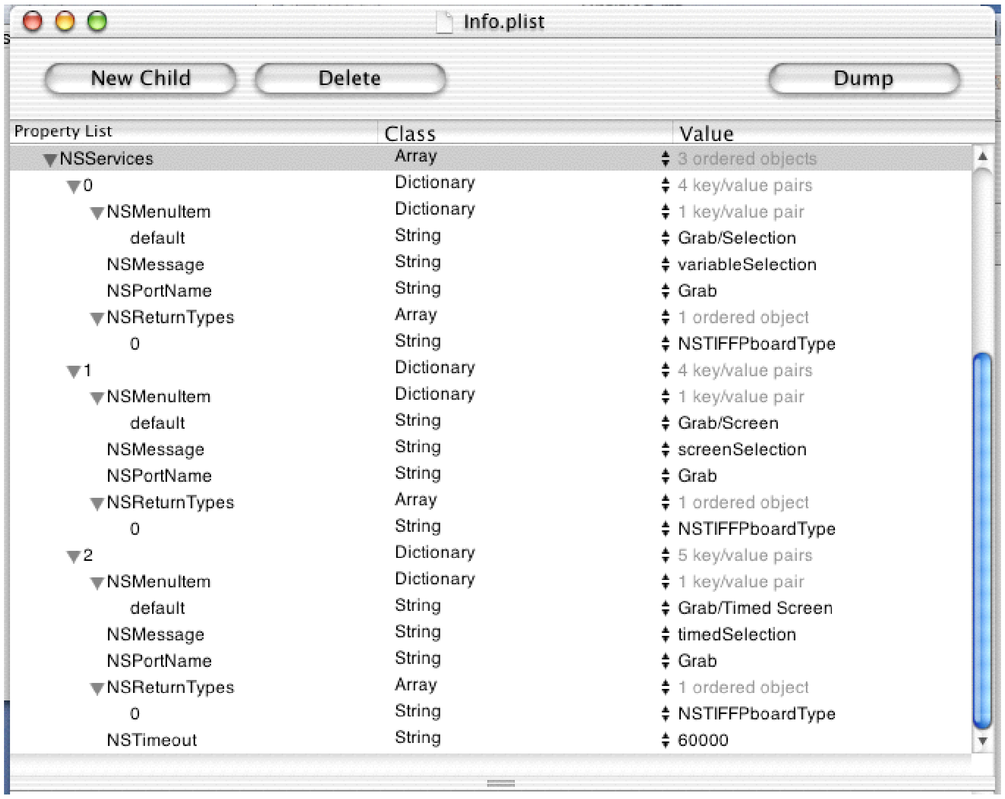
The `NSServices` property has three entries, one for each service offered by Grab. The first entry is for the menu item Grab > Selection. The slash notation—Grab/Selection—specifies that Selection should be an item in the Grab submenu. (See Figure 2-3 (page 10).)

Note that for each of the three entries, the port name is Grab. As mentioned, the port name is usually the application name.

Each entry has one return type, `NSTIFFPboardType`. An application could have more than one return type per entry, and the return types don't necessarily need to be the same for each entry.

The entry for Grab/Timed Screen is the only entry that has a specified timeout value. This optional entry is needed in this case so that the Grab application can wait for the user to set up the screen before taking a screenshot.

Figure 1-5 The NSServices property for the Grab application



Property List	Class	Value
▼NSServices	Array	3 ordered objects
▼0	Dictionary	4 key/value pairs
▼NSMenuItem	Dictionary	1 key/value pair
default	String	Grab/Selection
NSMessage	String	variableSelection
NSPortName	String	Grab
▼NSReturnTypes	Array	1 ordered object
0	String	NSTIFFPboardType
▼1	Dictionary	4 key/value pairs
▼NSMenuItem	Dictionary	1 key/value pair
default	String	Grab/Screen
NSMessage	String	screenSelection
NSPortName	String	Grab
▼NSReturnTypes	Array	1 ordered object
0	String	NSTIFFPboardType
▼2	Dictionary	5 key/value pairs
▼NSMenuItem	Dictionary	1 key/value pair
default	String	Grab/Timed Screen
NSMessage	String	timedSelection
NSPortName	String	Grab
▼NSReturnTypes	Array	1 ordered object
0	String	NSTIFFPboardType
NSTimeout	String	60000

What Happens When a Service Is Invoked

This section provides an overview of what happens when a service is invoked from a Carbon application. We'll use a fictitious text editing application—BestTextEdit—to illustrate what happens when the user chooses Services from the application menu. In this example, a text document is open and active, and there are several paragraphs of text visible. The user has selected some of the text and wants to mail the selected text to a colleague. Now the user opens the application menu and chooses Services.

As soon as the user chooses Services, the system sends a Carbon event of class `kEventClassService` and kind `kEventServiceGetTypes` to the application. In essence, the system wants to know what types the application (BestTextEdit) can provide (copy data types) or accept (paste data types). When the BestTextEdit application receives the Carbon event, it must determine what types are appropriate for the current state of the application. Recall the user has selected text (`OSType 'TEXT'`), so BestTextEdit should indicate it can provide text data. It should also be able to accept text data, and may be able to accept other types of data (such as TIFF).

The BestTextEdit application provides the data types in the Carbon event parameters `kEventParamServicesCopyTypes` and `kEventParamServicesPasteTypes`. Each parameter is of type `typeCFMutableArrayRef`, so more than one data type can be provided if it's appropriate.

Once the system knows what data types the BestTextEdit application can handle, it enables the appropriate services. At that point, the user can choose GreatMailApp> Mail To from the Services menu. Figuring out the data types and enabling the appropriate items in the Services menu happens quite rapidly. In fact, to the user, items should appear in the Services menu instantaneously.

When the user chooses GreatMailApp > Mail To from the Services menu, a Carbon event of class `kEventClassService` and kind `kEventServiceCopy` is sent to the BestTextEdit application. The BestTextEdit applicaton must then copy the current text selection to the scrap provided to the application in the services event. Once the data is copied, the Carbon Event Manager sends a Carbon event of class `kEventClassService` and kind `kEventServicePerform` to the GreatMailApp application.

The GreatMailApp application can provide two services: Mail To and Mail Text. Before mail does anything, it must determine which service it needs to provide to the BestTextEdit application. GreatMailApp does this by checking the Carbon event parameter `kEventParamServiceMessageName`. The string provided in this parameter specifies which service was invoked from the Services menu. When GreatMailApp determines Mail To is the service it must provide, the GreatMailApp application gets the text from the scrap provided by the Carbon Event Manager, opens a new message, and pastes the text into the new message.

What if nothing is selected in the document when the user chooses Services? The application can't provide any data, so BestTextEdit shouldn't provide any data types for the Carbon event parameter `kEventParameterServiceCopyTypes`. The items in the Services menu are dimmed and not available to the user.

If the user chooses a service that provides data, such as a screen capture service, the BestTextEdit application receives a Carbon event of class `kEventClassService` and kind `kEventServicePaste` after the service has copied the data to the scrap provided to the service by the Carbon Event Manager. The BestTextEdit application responds to the event by copying the data from the scrap and then handling it as appropriate, such as pasting it into the active document at the current insertion point.

Note: The Carbon Event Manager uses a specific scrap to pass data between a service provider and a service requestor. The notion of a specific scrap is new to Carbon. In the past there was one public scrap, what users refer to as the Clipboard, and what the Scrap Manager referred to as the current scrap, or simply the scrap. The Scrap Manager has a new function, `ClearScrap`, that takes a scrap reference as a parameter.

Specific scraps are parallel in concept to the Cocoa `NSPasteboard` class. Application services are actually implemented in Cocoa. Elsewhere in this document you'll see that other Cocoa concepts have been leveraged for the Carbon implementation of application services. (For example, see "Services Properties" (page 12).)

Application Services Tasks

This chapter shows you how add Services functionality to a Carbon application. The tasks covered in this chapter include:

- "Using a Service" (page 17). Follow the steps in this section if you want to set up your application to use services provided by other applications.
- "Providing a Service" (page 14). Read this section if you want to set up your application to provide one or more services to other applications.

Using a Service

Mac OS X version 10.1 and later automatically adds a Services item to the application menu of a Carbon application. If you do nothing else, the items in your application's Services submenu are dimmed and unavailable to users. If you want to enable the Services items appropriate to your application, you'll need to set up your application to handle the appropriate services-related Carbon events. Specifically, you need to do the tasks described in the following sections:

1. "Declaring Services Events" (page 17). You must declare the services-related Carbon events your application handles.
2. "Specifying the Data Types Used by Your Application" (page 35).
3. "Handling Copy and Paste Events" (page 20). You need to write a Carbon event handler that responds to each services-related Carbon event your application handles.
4. "Installing a Carbon Event Handler" (page 35).

Declaring Services Events

Services-related Carbon events are of event class `kEventClassService` and can be one of four event kinds, only three of which are relevant to applications that use, but do not provide, services.

- `kEventServiceGetTypes`. Any application that wants to use a service must respond to this event. When you receive this event, you must provide the data types that can be pasted into or cut from your application. The operating system enables items in the Services menu for those services that operate on, or provide, those data types.
- `kEventServiceCopy`. A service that operates on data supplied by your application (such as the Mail To service provided by the Mail application) sends this kind of event to your application. Your application needs to respond to this event by copying the currently selected data to the scrap provided by the Carbon Event Manager.

- `kEventServicePaste`. A service that provides data to your application (such as the Grab Screen service provided by the Grab application) sends this kind of event to your application. Your application needs to respond to this event by copying the data on the scrap provided by the Carbon Event Manager to your application.

A Carbon event type consists of an event class and an event kind that define an event. You use a structure of type `EventTypeSpec` to declare the Carbon events your application handles. Listing 3-1 (page 18) shows how you'd declare an `EventTypeSpec` structure for your application.

Listing 2-1 Declaring an EventTypeSpec structure for services events

```
const EventTypeSpec events[] ={
        { kEventClassService, kEventServiceGetTypes },
        { kEventClassService, kEventServiceCopy },
        { kEventClassService, kEventServicePaste }
}
```

It's more likely that your application will be handling other Carbon events, in which case you'd just add the services-related Carbon event types to the `EventTypeSpec` structure you set up for your entire application, similar to what is shown in Listing 3-2 (page 18).

Listing 2-2 Declaring an EventTypeSpec structure for a variety of Carbon events

```
const EventTypeSpec events[] ={
        { kEventClassService, kEventServiceGetTypes },
        { kEventClassService, kEventServiceCopy },
        { kEventClassService, kEventServicePaste },
        { kEventClassKeyboard, kEventRawKeyDown },
        { kEventClassWindow, kEventWindowClickContentRgn },
        { kEventClassWindow, kEventWindowFocusAcquired },
        { kEventClassWindow, kEventWindowFocusRelinquish },
        { kEventClassWindow, kEventWindowBoundsChanging },
        { kEventClassWindow, kEventWindowBoundsChanged }
}
```

Specifying the Data Types Used by Your Application

When the user chooses Services from your application menu, the system sends a Carbon event of class `kEventClassService` and kind `kEventServiceGetTypes` to your application. You need to respond to this event by specifying the data types used by your application. One approach is to declare the data types as items in an array, similar to what's shown in Listing 3-3 (page 18).

Listing 2-3 Declaring the data types used in an application

```
static const OSType MyAppsDataTypes[] =
{
    'TEXT',
    'PICT',
    'MooV',
    'AIFF',
    'utxt'
};
```

When your application receives the service event of kind `kEventServiceGetTypes`, you can append the items from the array you declared to the arrays passed to your application in the Carbon event parameters `kEventParamServiceCopyTypes` and `kEventParamServicePasteTypes`. Listing 3-4 (page 19) show how to get the Carbon event parameters provided by the event kind `kEventServiceGetTypes` and then append your application's data types to each of the arrays. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-4 Supplying the data types used by your application

```
Boolean             textSelection = !TXNIsSelectionEmpty (myTextObject);       // 1
CFMutableArrayRef   copyTypes, pasteTypes;
short               index, count;
if (textSelection)
{
    GetEventParameter (inEvent,
                        kEventParamServiceCopyTypes,
                        typeCFMutableArrayRef,
                        NULL,
                        sizeof (CFMutableArrayRef),
                        NULL,
                        &copyTypes);                                           // 2
}
GetEventParameter (inEvent,
                    kEventParamServicePasteTypes,
                    typeCFMutableArrayRef,
                    NULL,
                    sizeof (CFMutableArrayRef),
                    NULL,
                    &pasteTypes);                                              // 3
count = sizeof (MyAppsDataTypes) / sizeof (OSType);                            // 4
for ( index = 0; index < count; index++ )
{
    CFStringRef type =
                CreateTypeStringWithOSType (MyAppsDataTypes [index]);          // 5
    if (type)
    {
        if (textSelection) CFArrayAppendValue (copyTypes, type);              // 6
        CFArrayAppendValue (pasteTypes, type);                                 // 7
        CFRelease (type);                                                      // 8
    }
}
return noErr;                                                                  // 9
```

Here's what the code does:

1. Check to see whether there is any text selected. If there isn't, then you won't need to provide data types for copying, as there won't be anything to copy. You can use whatever function is appropriate to check for selected text. The Multilingual Text Engine (MLTE) function `TXNIsSelectionEmpty` returns a Boolean value that indicates whether or not anything is selected in an MLTE text object (`TXNObject`).

2. If text is selected, then use the Carbon Event Manager function `GetEventParameter` to get the array associated with the Carbon event parameter `kEventParamServiceCopyTypes`. You'll fill this array with the data types the application can give to a service.

3. Use the Carbon Event Manager function `GetEventParameter` to get the array associated with the Carbon event parameter `kEventParamServicePasteTypes`. You'll fill this array with the data types your application can accept from a service.

4. Recall that the array `MyAppsDataTypes` was declared in Listing 3-3 (page 18). This line of code figures out how many data types are in the array.

5. Use the Carbon Event Manager function `CreateTypeStringWithOSType` to create a `CFStringRef` object from the `OSType`.

   Before your application can use services provided by other applications, your application must provide the data types it can handle to the Carbon Event Manager. The copy and paste arrays passed to your application by the Carbon Event Manager are arrays of `CFStringRef` objects. As a result, you must convert the `OSType` data types your application handles to the equivalent `CFStringRef` objects using the function `CreateTypeStringWithOSType`.

6. If text is selected, use the Core Foundation Collection Services function `CFArrayAppendValue` to add the data types that the application uses to the `copyTypes` array. This statement is in a `for` loop, so all the data types are appended by the time the loop is completed.

7. This statement is also in the `for` loop. Again, use the function `CFArrayAppendValue` to add the data types the application uses to the `pasteTypes` array.

8. Call the Core Foundation Base Services function `CFRelease` to release the `type` object.

9. Returns `noErr` to indicate to the event dispatching system that the event has been handled. If you return anything else, the services request does not complete because the system assumes that you are unable to handle this event.

Handling Copy and Paste Events

After you've specified which data types your application handles, your application can receive the service events of kind `kEventServiceCopy` and `kEventServicePaste`. You need to respond to each event kind appropriately.

This section shows code segments that handle each event kind. The code segments are from a text editing application that uses Multilingual Text Engine (MLTE) to handle Unicode text editing. Your code needs to be tailored for your application.

Handling a Copy Event

Listing 3-5 (page 20) shows a code segment from an event handler that handles the event kind `kEventServiceCopy`. Before you handle a copy event you should make sure there is data selected. You don't need to do anything unless some data is selected. A detailed explanation for each numbered line of code follows the listing.

Listing 2-5 Handling a copy event

```
case kEventServiceCopy:
{
    OSStatus        status = noErr;
    TXNOffset       startOffset;
    TXNOffset       endOffset;
    ScrapRef        scrap;
    Handle          textHandle;

    check (myTextObject);                                                   // 1
    check (!TXNIsSelectionEmpty (myTextObject));                            // 2

    TXNGetSelection (myTextObject, &startOffset, &endOffset);               // 3

    status = TXNGetData (myTextObject,
                        startOffset, endOffset, &textHandle);               // 4
    require_noerr (status, CantGetTXNData);                                 // 5
    require (textHandle != NULL, CantGetDataHandle);                        // 6

    status = GetEventParameter (myEvent,
                                kEventParamScrapRef,
                                typeScrapRef,
                                NULL,
                                sizeof (ScrapRef),
                                NULL,
                                &scrap );                                   // 7
    require_noerr (status, CantGetCopyScrap);                               // 8
    verify_noerr (ClearScrap (&scrap));                                     // 9
    status = PutScrapFlavor (scrap,
                            kTXNUnicodeTextData,
                            0,
                            GetHandleSize (textHandle),
                            *textHandle );                                  // 10
CantGetCopyScrap:
    DisposeHandle (textHandle);                                             // 11
CantGetDataHandle:
CantGetTXNData:
    result = status;                                                        // 12
}
break;
```

Here's what the code does:

1. Calls the macro `check` to make sure the text object is valid. The variable `myTextObject` is of type `TXNObject`. See the MLTE reference for more information. See Debugging.h for more information on the `check` macro.

2. Calls the MLTE function `TXNIsSelectionEmpty` to make sure there is something in the selection, otherwise it doesn't make sense to copy.

3. Gets the absolute offsets of the selected text by calling the MLTE function `TXNGetSelection`.

4. Calls the MLTE function `TXNGetData` to copy the selected text to a text handle.

5. Checks to make sure the function `TXNGetData` did not return an error. See Debugging.h for more information on the `require_noerr` macro.

6. Checks to make sure the text handle returned by the function `TXNGetData` is not `NULL`. See Debugging.h for more information on the `require` macro.

7. Calls the Carbon Event Manager function `GetEventParameter` to obtain the scrap associated with the copy event.

8. Calls the macro `require_noerr` to make sure the parameter is obtained without error.

9. Clears the scrap associated with the event, also calling the macro `verify_noerr` to check for errors. Clearing the scrap ensures you can safely copy data to it. See Debugging.h for more information on the `verify_noerr` macro.

10. Calls the Scrap Manager function `PutScrapFlavor` to place the selected text on the scrap.

11. Disposes of the text handle previously allocated by the function `TXNGetData`.

12. Sets `result` to `status`. This code listing does not show the complete event handling routine; only the switch statement that handles a copy event. Note that the event handler must return the `result` value. The value `noErr` indicates to the event dispatching system that the event has been handled. If you return anything else, the services request does not complete because the system assumes that you are unable to handle this event.

Handling a Paste Event

Listing 3-6 (page 22) shows a code segment from an event handler that handles the event kind `kEventServicePaste`. Unlike the copy event, you do not need to check for selected data, as your application is accepting data provided by a service. A detailed explanation for each numbered line of code follows the listing.

Listing 2-6 Handling a paste event

```
case kEventServicePaste:
{
    OSStatus    status = noErr;
    Size        unicodeTextSize;
    UniChar*    unicodeText;
    ScrapRef    scrap;
    TXNOffset   startOffset;
    TXNOffset   endOffset;

    check (myTextObject != NULL);

    status = GetEventParameter (myEvent,
                        kEventParamScrapRef,
                        typeScrapRef,
                        NULL,
                        sizeof (ScrapRef),
                        NULL,
                        &scrap);                                // 1
    require_noerr (status, CantGetPasteScrap );                 // 2

    status = GetScrapFlavorSize (scrap, kTXNUnicodeTextData,
                            &unicodeTextSize);                  // 3
    require_noerr (status, CantGetDataSize);                    // 4

    unicodeText = (UniChar*) malloc (unicodeTextSize);          // 5
    require_action (unicodeText != NULL, CantCreateBuffer,
                                status = memFullErr);                    // 6

    status = GetScrapFlavorData (scrap, kTXNUnicodeTextData,
                                &unicodeTextSize, unicodeText );         // 7
    require_noerr (status, CantGetScrapFlavorData);                      // 8

    TXNGetSelection (myTextObject, &startOffset, &endOffset);            // 9

    status = TXNSetData (myTextObject, kTXNTextData,
                    unicodeText, unicodeTextSize,
                    startOffset, endOffset );                            // 10

CantGetScrapFlavorData:
    free (unicodeText);                                                  // 11

CantCreateBuffer:
CantGetDataSize:
CantGetPasteScrap:
    result = status;                                                     // 12
}
```

Here's what the code does:

1. Calls the Carbon Event Manager function `GetEventParameter` to get the scrap associated with the paste event. This scrap contains data provided by a service.
2. Calls the macro `require` to make sure the parameter contains data. See Debugging.h for more information on the `require` macro.
3. Gets the size of the data on the scrap by calling the Scrap Manager function `GetScrapFlavorSize`.
4. Calls the macro `require_noerr` to make sure the size of the data on the scrap is obtained without error. See Debugging.h for more information on the `require_noerr` macro.
5. Allocates a Unicode text buffer large enough to hold the data on the scrap.
6. Calls the macro `require_action` to make sure the Unicode text buffer is not `NULL`. See Debugging.h for more information on the `require_action` macro.
7. Obtains the data on the scrap by calling the Scrap Manager function `GetScrapFlavorData`.
8. Calls the macro `require_noerr` to make sure the data is obtained without error.
9. Calls the MLTE function `TXNGetSelection` to obtain the starting and ending offsets that define the range into which the data should be pasted.
10. Pastes the data from the scrap to the MLTE text object (`TXNObject`) by calling the MLTE function `TXNSetData`.
11. Frees the memory allocated for the Unicode text buffer.
12. Sets `result` to `status`. This code listing does not show the complete event handling routine; only the switch statement that handles a paste event. Note that the event handler must return the `result` value. The value `noErr` indicates to the event dispatching system that the event has been handled. If you return anything else, the services request does not complete because the system assumes that you are unable to handle this event.

Installing a Carbon Event Handler

Once you've written a handler to take care of the services events, you need to install it by calling the Carbon Event Manager function `InstallEventHandler` with the appropriate event target. You need to get the event target before you call the function `InstallEventHandler`. For the text editing application used in the sample code, the event target is the document window. You can use code similar to the following to get a window event target:

```
myEventTarget = GetWindowEventTarget (myDocumentWindow);
```

To install the event handler, your application should include a call similar to this:

```
InstallEventHandler (myEventTarget,
                        handlerUPP,
                        numTypes,
                        typeList,
                        userData,
                        &handlerRef);
```

where:

- `myEventTarget` is an event target reference.
- `handlerUPP` is a pointer to your handler function.
- `numTypes` is the number of events for which you are registering the handler.
- `typeList` is a pointer to the array of events that contains the services events.
- `userData` is an optional value that is passed to your event handler when the handler is called.
- `handlerRef` is (on return) a value of type `EventHandlerRef`, which you can use later to remove the handler. You can pass `null` if you don't want the reference. When the target is disposed of, the handler is removed as well.

Providing a Service

To provide one or more services for other applications to use, you need to do the tasks described in the following sections

1. "Adding the NSServices Property to the Information Property List" (page 25). Your application advertises the services it provides through the `NSServices` property.
2. "Making Sure There's a Bundle Identifier Property" (page 26). The `CFBundleIdentifier` property should already be in your application's property list. If it isn't, you need to add it.
3. "Handling the Service Perform Event" (page 26). You must write code to handle the Carbon event kind `kEventServicePerform`.
4. "Installing a Carbon Event Handler" (page 32). The event target for the handler should be the application.
5. "Registering the Service" (page 33). Registration informs the system and other applications know about your service.

Adding the NSServices Property to the Information Property List

Applications must use their information property list to advertise the services they provide. You need to add the `NSServices` property, and it must have one dictionary entry for each service you provide. Each dictionary entry must have these keys: `NSMessage`, `NSPortName`, `NSMenuItem`, `NSSendTypes`, and `NSReturnTypes` and can optionally have these keys: `NSKeyEquivalent`, `NSUserData`, and `NSTimeout`. See "Services Properties" (page 12) for a description of each key and examples of the `NSServices` property in an information property list.

Services functionality in Mac OS X is implemented using Cocoa. This has implications for the values you assign to the `NSSendTypes` and `NSReturnTypes` arrays. Cocoa applications use `NSPasteboard` objects as interfaces to a pasteboard server that allows you to transfer data between applications, as in copy, cut, and paste operations. As a result, Cocoa applications can recognize only those services whose `NSSendTypes` and `NSReturnTypes` arrays contain `NSPasteboard` types.

Services is implemented to allow Carbon applications to recognize both `OSType` and `NSPasteboard` types. If you assign only `OSType` types to the `NSSendTypes` and `NSReturnTypes` arrays, Cocoa applications aren't able to access your services.

Note: To make your services available to all applications, you should assign only `NSPasteboard` types (such as `NSStringPboardType`) to the `NSSendTypes` and `NSReturnTypes` arrays.

To add the `NSServices` property to the information property list for a service application or a standalone service do the following:

1. Open your services application or standalone services project in Project Builder.
2. Click the Targets tab, then click the appropriate target in the Targets list.
3. Click the Application Settings tab, then click Expert.
4. Click the New Sibling button.
5. Type `NSServices` in the Property List column.
6. Choose Array from the Class pop-up menu.
7. Click the disclosure triangle next to `NSServices`, then click the New Child button.

   When you click the disclosure triangle, the New Sibling button changes to New Child.
8. Set the array element's class to Dictionary.
9. Click the disclosure triangle next to the array element, then click the New Child button.
10. Type an `NSServices` keyword in the Property List column, make sure its class is set appropriately, then type or choose a value.

    Click New Sibling to add the other required keywords to this array element.

    See "Services Properties" (page 12) for a discussion of keywords and their classes.

    For each service you provide, you need to specify a string for the `NSMessage` property. You'll use this string to determine which service you need to provide.

You need to add an array element to the `NSServices` property for each service your application provides. To add another array element, click `NSServices`, click the New Child button, then follow steps 8 through 10.

Making Sure There's a Bundle Identifier Property

The `CFBundleIdentifier` property is a unique identifier string for a bundle (such as an application bundle). All bundles created for Mac OS X should have this property in the information property list, as the system uses the identifier to locate the bundle at runtime and find out information about the bundle, such as what services are provided.

If you haven't already added a bundle identifier to the information property list, do so now. The value of the `CFBundleIdentifier` property should be a string in the form of a Java-style package name (think of it as a reverse URL). For example, `com.mycompany.myGreatTextEditApp`.

Handling the Service Perform Event

The Carbon Event Manager requests a service by passing your application a services event of kind `kEventServicePerform`. You must do the following to handle this event:

1. Declare an `EventTypeSpec` structure that specifies the Carbon event class `kEventClassService` and event kind `kEventServicePerform`. See "Declaring Services Events" (page 17) for information on declaring an `EventTypeSpec` structure.

2. Write code to handle the event. The details for handling the event are in this section.

When your application receives the event kind `kEventServicePerform`, you need to get the Carbon event parameters `kEventParamServiceMessageName`, `kEventParamScrapRef`, and (optionally) `kEventParamUserData` as shown in Listing 3-7 (page 26). You also need to set up some variables to receive the event parameters and to handle the scrap used to pass data between the service requestor and your service. A detailed explanation for the code in Listing 3-7 (page 26) follows the listing.

Listing 2-7 Getting the Carbon event parameter for the event kind kEventServicePerform

```
CFStringRef     message;                                                    // 1
CFStringRef     userData;                                                   // 2
ScrapRef        specificScrap;                                              // 3
ScrapRef        currentScrap                                                // 4

GetEventParameter (inEvent,
                kEventParamServiceMessageName,
                typeCFStringRef,
                NULL,
                sizeof (CFStringRef),
                NULL,
                &message);                                                  // 5
GetEventParameter (inEvent,
                kEventParamScrapRef,
                typeScrapRef,
                NULL,
                sizeof (ScrapRef),
                NULL,
                &specificScrap);                                // 6
GetEventParameter (inEvent,
                kEventParamUserData,
                typeCFStringRef,
                NULL,
                sizeof (CFStringRef),
                NULL,
                &userData);                                     // 7
```

Here's what the code does:

1. Declare a variable to hold the `NSMessage` string that gets returned when you call the function `GetEventParameter` to get the Carbon event parameter `kEventParamServiceMessageName`. Recall that `NSMessage` is a property whose value identifies a service provided by your application. You use the string returned in the message `variable` to determine which service to invoke in response to the Carbon event. For example, if the Grab application gets the `timedSelection` string, it would call its function for taking a shot of the screen after a specified amount of time.

2. Declare the variable `userData` only if you've added an `NSUserData` property for the services offered by your application.

3. Declare a variable to hold the scrap passed by the Carbon Event Manager. The notion of having specific scraps in Carbon is new with Mac OS X version 10.1. You'll write code later that moves data from this scrap to the Clipboard or the MLTE private scrap.

4. Declare a variable to hold the current scrap, otherwise known as the Clipboard.

5. Call the Carbon Event Manager function `GetEventParameter` to get the name of the service. It's the string you assigned to the `NSMessage` property, and the string is returned in the `message` variable.

6. Get the scrap reference from the Carbon Event Manager. If the service requestor is providing data to your service, the scrap contains data. Otherwise, this is the scrap you use to pass data to the service requestor.

7. If you assigned a string to `NSUserData`, you need to call the function `GetEventParameter` to get the `userData` string.

Once you retrieve the values of the `message` and the `scrap` variables, you can invoke the code to handle the service specified by the `message`. Next we'll take a look at how to implement the code to handle a service.

When you write the code to handle a service, you need to determine how data moves between the application requesting a service and your service. The way data moves has implications for clearing the scrap and moving data between the specific scrap provided by the Carbon Event Manager, the current scrap (that is, the Clipboard), and, in the case of a text editing application, the MLTE private scrap. There are three possible scenarios.

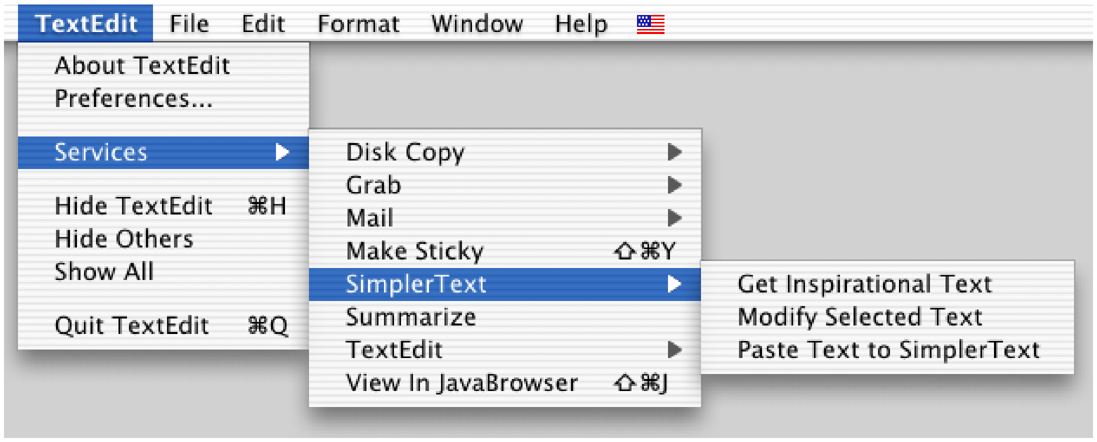
1. Data is given to the service provider by the service requestor, but data is not returned to the service requestor.

2. Data is given to the service requestor by the service provider. The service requestor doesn't give any data to the service provider.

3. Data is given to the service provider by the service requestor, and the service provider returns data to the service requestor.

Figure 3-1 (page 28) shows three services provided by the SimplerText application, a sample text editing application that uses MLTE. The Paste Text to Simpler Text service illustrates the first scenario. Get Inspirational Text illustrates the second scenario. Modify Selected Text illustrates the third scenario. The code segments in the following sections show how each service is implemented.

Figure 2-1 Three services provided by SimplerText



Data is Given to the Service Provider

Data is given to the service provider on the scrap and none is returned to the service requestor. The data is extracted from the scrap by the service provider and the provider does something with it. For example, the Stickies application provides a Make Sticky service that takes the current selection from an application, and pastes it into a new Sticky document.

Your application, as the service provider, must call the function `ClearCurrentScrap` to clear the current scrap before putting data on it. Then you need to call the function `GetCurrentScrap` to get a valid scrap reference. Listing 3-8 (page 28) show a code segment that implements the Paste Text to Simpler Text service. It gets text data from a service requestor and pastes it into a document. See the detailed explanation that follows the listing for each numbered line of code.

Listing 2-8 Getting data from the service requestor

```
ClearCurrentScrap ();                                                   // 1
GetCurrentScrap (&currentScrap);                                        // 2
count = sizeof (MyAppsDataTypes) / sizeof (OSType);                     // 3
for ( index = 0; index < count; index++ )                               // 4
{
        Size        byteCount;
        OSStatus    err;

        err = GetScrapFlavorSize (specificScrap,
                                  MyAppsDataTypes [index],
                                  &byteCount);
        if ( err == noErr )
       {
            void*   buffer = malloc (byteCount);

           if (buffer != NULL)
           {
                err = GetScrapFlavorData (specificScrap,
                                    TXNTypes [index],
                                    &byteCount,
                                    buffer);
                if (err == noErr)
                {
                       PutScrapFlavor (currentScrap,
                                TXNTypes [index],
                                0,
                                byteCount,
                                buffer);
                }
                free (buffer);
            }
        }
}
TXNConvertFromPublicScrap ();                                                    // 5
if (TXNIsScrapPastable())
            TXNPaste (myTextObject);                                             // 6
```

Here's what the code does:

1. When your application receives a request to copy text from the service requestor into your application, you must first clear the current scrap using the Scrap Manager function `ClearCurrentScrap`.

2. Get a reference to the current scrap.

3. `MyAppsDataTypes` is an array that contains the `OSTypes` handled by your application. You need to declare this array. See Listing 3-3 (page 18) for an example.

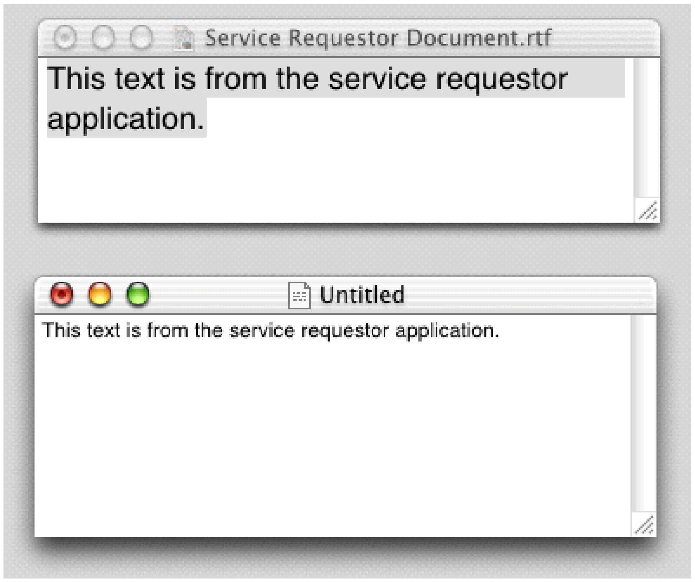
4. The `for` loop copies the data from the scrap provided by the Carbon Event Manager. For each data type your application can handle, you call the function `GetScrapFlavorSize` to see if the type of data on the scrap provided by the CEM matches. If it matches, allocate a buffer and get the data from the scrap using the function `GetScrapFlavorData`. Then use the Scrap Manager function `PutScrapFlavor` to copy the data to the Clipboard.

5. MLTE uses a private scrap, so you need to call the MLTE function to convert the current scrap to MLTE's private scrap.

6. Make sure the data on MLTE's scrap can be pasted. If so, then paste it into your document.

Figure 3-2 (page 30) illustrates what the code in Listing 3-8 (page 28) does. The text provided by the service requestor is highlighted in the service requestor document window. The service provider gets the text from the scrap and pastes it into an untitled document.

Figure 2-2 Text provided by the service requestor



This text is from the service requestor application.

This text is from the service requestor application.

Data is Given to the Service Requestor

Data is given to the service requestor by the service provider, but the service requestor doesn't give any data to the service provider. For example, the Grab application provides a Grab Screen service that gives the client an image of the screen.

Your application, as the service provider, must call the Scrap Manager function `ClearScrap` to clear the scrap given to it by the Carbon Event Manager. Then it can put data on the scrap for the service requestor by calling the function `PutScrapFlavor`. Listing 3-9 (page 30) shows a code segment that implements the Get Inspirational Text service. It gives a text string to the service requestor. See the detailed explanation that follows the listing for each numbered line of code.

Listing 2-9 Giving data to the service requestor

```
char    string[] = "Buy low, sell high.";                    // 1

ClearScrap (&specificScrap);                                 // 2
PutScrapFlavor (specificScrap,
            'TEXT',
            0,
            strlen (string),
            string);                                         // 3
```

Here's what the code does:

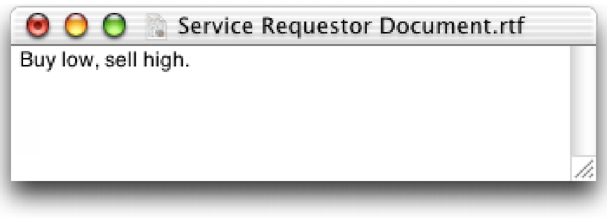
1. Declare a string. This is what you'll give to the service requestor.

2. The function `ClearScrap` is new with Mac OS X version 10.1. Unlike the function `ClearCurrentScrap`, which can only clear the Clipboard, `ClearScrap` can clear any scrap passed to it. In this case, you need to pass a reference to the scrap provided to your application by the Carbon Event Manager in the parameter `kEventParamScrapRef`.

3. After you've cleared the scrap, put the string you want to pass to the service requestor on the scrap.

Figure 3-3 (page 31) illustrates what the code in Listing 3-9 (page 30) does. The string defined in the listing is pasted into the service requestor's document window.

Figure 2-3 Giving text to the service requestor



Buy low, sell high.

Data is Given to the Service Provider; Data is Returned to the Service Requestor

The service requestor gives data to the service provider. The service provider modifies the data and returns the modified data to the service requestor. For example, a spell checker service takes text from the service requestor and returns text with corrected spelling.

Your application, as the service provider, must use the function `GetScrapFlavorData` to take the data from the scrap given by to it by the Carbon Event Manager. Then the scrap must be cleared, by calling the Scrap Manager function `ClearScrap`. Once the data is processed, you put it on the scrap by calling the function `PutScrapFlavor`.

Listing 3-10 (page 31) shows a code segment that implements the Modify Selected Text service. It gets text data from a service requestor, modifies it, and passes it back to the service requestor. See the detailed explanation that follows the listing for each numbered line of code.

Listing 2-10 Modifying data from the service requestor

```
OSStatus    err;                                                      // 1
Size        byteCount;

err = GetScrapFlavorSize (specficScrap,
                          'utxt',
                          &byteCount);                                // 2
if (err == noErr)
{
    UniChar*    buffer = (UniChar*) malloc (byteCount);               // 3
    if (buffer != NULL)
    {
        err = GetScrapFlavorData (specificScrap,
                                  'utxt',
                                  &byteCount,
                                  buffer);                            // 4
        if ( err == noErr && byteCount > 1 )
        {
            buffer[0] = '!';                                          // 5
            ClearScrap (&specficScrap);                               // 6
            PutScrapFlavor (specificScrap,
                            'utxt',
                            0,
                            byteCount,
                            buffer);                                  // 7
            result = noErr;
        }
        free (buffer);                                                  // 8
    }
}
```

Here's what the code does:

1. Declare variables for error checking and to get the number of bytes on the scrap.

2. Find out the size of the Unicode text data that's on the scrap provided by the Carbon Event Manager.

3. Allocate a buffer to accommodate the size of the data.

4. Get the data from the scrap.

5. Replace the first character with an exclamation point.

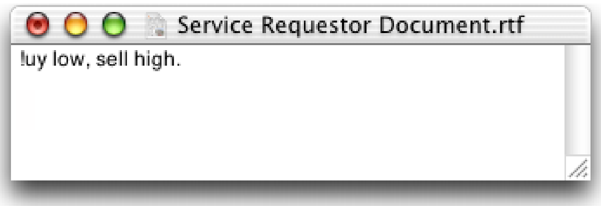
6. Call the Scrap Manager function `ClearScrap` to clear the scrap given to your application by the Carbon Event Manager. Recall that the function `ClearScrap` can clear any scrap passed to it, while the function `ClearCurrentScrap` is limited to clearing the Clipboard.

7. Put the modified data back on the scrap for the Carbon Event Manager.

8. Free the memory allocated for the buffer.

Figure 3-4 (page 32) illustrates what the code in Listing 3-10 (page 31) does. The original text was `Buy low, sell high`. The service provider replaced the first letter of the selected text with an exclamation point.

Figure 2-4 Modified text returned to the service requestor



!uy low, sell high.

Installing a Carbon Event Handler

Once you've written a handler to take care of the services events, you need to install it by calling the Carbon Event Manager macro `InstallApplicationEventHandler`. You can use this macro rather than the more general function `InstallEventHandler` when the event target is the application itself, which is the case for an application that provides services.

Your application should include a call similar to this:

```
InstallApplicationEventHandler ( handlerUPP,
                                 numTypes,
                                 typeList,
                                 userData,
                                 &handlerRef);
```

where:

- `handlerUPP` is a pointer to your handler function.
- `numTypes` is the number of events for which you are registering the handler.
- `typeList` is a pointer to the array of events that contains the services events.
- `userData` is an optional value that is passed to your event handler when the handler is called.
- `&handlerRef` is (on return) a value of type `EventHandlerRef`, which you can use later to remove the handler. You can pass `null` if you don't want the reference. When the target is disposed of, the handler is removed as well.

Registering the Service

If you want to make sure your service gets registered and your code works, you'll need to do the following.

- Install your application or standalone service in the appropriate location. See "Installation Paths for Services" (page 11).
- Log out and back in. Mac OS X searches for and assembles the list of available services only when a user logs in.

Carbon Events for Services

This chapter outlines the Carbon event classes, kinds, and parameters your application needs to install Carbon event handlers for services events. For more information on Carbon events, see the documentation for the Carbon Event Manager on the Carbon Developer Documentation website.

Event Classes and Kinds

There are four event kinds associated with the Carbon event class `kEventClassService`.

Table A-1 Event classes and kinds used by application services

Event Class	Event Kind	Means
`kEventClassService`	`kEventServiceCopy`	Copy event
	`kEventServicePaste`	Paste event
	`kEventServiceGetTypes`	Get data types handled by the calling application
	`kEventServicePerform`	An application is requesting a service

Event Parameters for Services Events

The Carbon event parameters that are available for a services event vary depending on the kind of the services event.

Table A-2 Carbon event parameters for service events

Event Kind	Event Parameter	Type
`kEventServiceGetTypes`	`kEventParamServiceCopyTypes`	`typeCFMutableArrayRef`
	`kEventParamServicePasteTypes`	`typeCFMutableArrayRef`
`kEventServiceCopy`	`kEventParamScrapRef`	`typeScrapRef`
`kEventServicePaste`	`kEventParamScrapRef`	`typeScrapRef`
`kEventServicePerform`	`kEventParamScrapRef`	`typeScrapRef`
	`kEventParamServiceMessageName`	`typeCFStringRef`

Event Kind	Event Parameter	Type
	`kEventParamServiceUserData`	`typeCFStringRef`

Document Revision History

This table describes the changes to *Setting Up Your Carbon Application to Use the Services Menu*.

Date	Notes
2003-12-10	Revised Listing 3-4 (page 19) so it returns `noErr`.
	Added a numbered comment to Listing 3-5 (page 20) and Listing 3-6 (page 22) that discusses why `noErr` needs to be returned by the event handler.
	Reorganized the introduction and updated hyperlinks.
2002-11-19	Revised sample code for handing copy and paste events.
	Removed the Appendix that discussed the Scrap Manager function `ClearScrap` and the Carbon Event Manager function `CreateTypeStringWithOSType`. These functions are documented in their respective managers.
2002-10-02	Fixed typographical and formatting errors.
2001-08-31	First release of this document.