Services

The NEXTSTEP services facility allows an application to make use of the services of other applications without knowing in advance what those services might be. For example, a text editing application lets the system know that it's willing to provide plain ASCII text or rich text (RTF) on the pasteboard any time there is a selection. Any service-providing application is then able to receive that text and act upon it. A service-providing application could thus provide such services as spell checking, grammar checking, encryption, reformatting, language translation, conversion to speech, or any number of useful functions. Service-providing applications can also place data back on the pasteboard to be received by the main application. In this way, data can be seamlessly exchanged between applications, and any application can extend the functionality of many others. This document provides a basic overview of the process of providing and using services.

Providing a Service

In order to provide a service, an application must make known the data types it's willing to act upon, the messages it must receive to initiate action, the menu item to be placed in applications that can provide or accept such data, and the Mach port on which it can receive the messages it published.

As an example, consider a service to reverse text. This service will accept ASCII text on the pasteboard, reverse it, and place the reversed ASCII text back on the pasteboard. Since the Text class supplied with NEXTSTEP knows how place and receive text on the pasteboard, all NEXTSTEP applications will be able to take advantage of the text reversal service to check palindromes or for simple encryption. Since the text will be automatically replaced in a Text object, it will be as though this feature were built in to every application.

First, you must declare the important aspects of the Reverser service in a text file looking something like this:

```
Message: reverseData
Port: Reverser
Send Type: NXAsciiPboardType
Return Type: NXAsciiPboardType
Menu Item: Reverse It
```

This is known as a service specification. (More than these five fields may be listed. The complete specification is described later.) For this example, we will call this file **services.text**. The **Send Type** field indicates that this service requires that data of NXAsciiPboardType be placed on the pasteboard. NXAsciiPboardType is a data type consisting of simple ASCII text. NEXTSTEP defines data types for simple and rich text, file names, encapsulated PostScript, TIFF image data and others. A service is free to request any data types it likes, including proprietary formats, but the service will only be enabled if the main application can supply data of that type. The **Return Type** field indicates that the service will return ASCII data on the pasteboard after manipulating the data. A return type isn't necessary; the service could simply act on the data it receives without returning anything to the main application. However, since this service returns data, the main application will wait for the service to provide data before it continues processing. (If the service doesn't return data, the service is invoked asynchronously and the main application doesn't wait.) Both the **Send Type** and **Return Type** fields are optional. A service may just accept data, it may just provide data, or it may modify data to be pasted back into the main application. A service may also list multiple send or return types indicating that it can accept any one of several types or that it will return many types; to indicate this, the **Send Type** or **Return Type** lines can be duplicated.

The **Menu Item** field indicates that a **Reverse It** command should be added to the Services menu of every application that can (at least under some circumstances) send and receive ASCII text. This

command will be enabled any time a text field has a selection that can be reversed. If the user chooses the enabled **Reverse It** command, the selection will be placed on the pasteboard and the message indicated in the **Message** field will be send to the port indicated in the **Port** field.

How a Service Is Advertised

NEXTSTEP uses the Mach-O executable file format, which effectively provides a simple directory structure to executable files. An executable thus contains multiple segments (akin to directories) each with a number of sections containing binary code, images, text, and other data. At run time, the system will look into the executable files in **~/Apps** and **/LocalApps**. If an executable contains a **_ _services** section in its **_ _ICON** segment, the services listed in the section will be made available to the appropriate applications. The following line must be added to the **Makefile.preamble** file (included by NEXTSTEP's standard makefile) to include the above **services.text** file:

```
LDFLAGS = -sectcreate _ _ICON _ _services services.text
```

How to Implement a Service

The message line in the **services.text** file indicates that the system will send a **reverseData** message to the service-providing application when the user clicks its menu item. Before sending such a message, the system will ask the main application to put the selected data on the pasteboard in the format required by the service. (The service's menu item will only be enabled when the main application has confirmed that it will be able to provide the data; more on this later.) The actual message sent to the service provider contains parameters identifying the pasteboard, supplying optional information about which service is actually to be performed (since a single method can be used to perform multiple services), and a pointer allowing the service to return an error message. Here is a possible implementation of the service to reverse text:

```
- reverseData: (id)pasteboard
    userData:(const char *)userData
    error:(char **)msg
{
    const char *types[1];
    char *buffer, *revBuffer, *data;
    int length, i=0, j;

   [pasteboard types];    // pretend to check the pasteboard types

   // read the ASCII data from the pasteboard
    if ([pasteboard readType:NXAsciiPboardType data:&data
        length:&length])
    {
        buffer = malloc(length+1);
        revBuffer = malloc(length+1);

       strncpy(buffer,data,length);
        buffer[length]='\0';
        revBuffer[length]='\0';

       // Reverse the text into revBuffer
        j = length - 1;
        while (j >=0) revBuffer[i++] = buffer[j--];

       // Write the reversed buffer back to the pasteboard
        types[0] = NXAsciiPboardType;
        [pasteboard declareTypes: types num: 1 owner:nil];
        [pasteboard writeType: NXAsciiPboardType data:revBuffer
            length:length];

       free(buffer);
        free(revBuffer);
```

```
    }
     else *msg = "Error: couldn't reverse text.";

     return self;
}
```

Every application has a Listener object to receive Objective C messages from external applications. This Listener registers its Mach port with the network name server under the application's name (Reverser, in this case), and it's used to receive messages from the Workspace Manager to open files or to receive notification that the system will shut down. This same Listener can also be used to receive messages from applications requesting services.

The Listener must be told which object within the application implements the methods that respond to service requests. This object is referred to as the *services delegate*. For example, to inform the application's Listener object that *theServiceObject* is the services delegate, you'd send these messages:

```
id theListener = [NXApp appListener];
[theListener setServicesDelegate:theServiceObject];
```

Thereafter, the Reverser application will receive the **reverseData:userData:error:** message any time the user requests that the selected text be reversed.

Fields in a Service Specification

The following fields must be included in a service specification:

Message: *<name of message>*

This field identifies the method that will be invoked in the Listener's service delegate. In the example above, the message field is **reverseData**, so the delegate must implement a **reverseData:userData:error:** method.

Port: *<name of a port>*

The name of a port of a Listener that is listening for service messages. Since every NEXTSTEP application has, by default, a Listener port registered under the application's name, the sevice specification generally uses this port name.

Menu Item: *<string appearing in other app's Services Menu>*

The menu item for the service. This string will appear in the Services menus of applications that can take advantage of the service. If this string contains a '/' character, that character will be used as a delimiter to specify a second level in the Service menu hierarchy. For example, a menu item of **Encrypt/Replace** will create a submenu **Encrypt** in the Services menu, with a **Replace** menu item in that submenu. Only one level of hierarchy is supported.

The menu item field must be untranslated. Translated menu items can be achieved by including menu item fields preceded by the appropriate language; for example **French Menu Item**. Alternatively, the supplied string can be used as a key into a ª.stringsº file (see **NXStringTable()**) called ***Language*.lproj/ServicesMenu.strings** found in the same directory as the executable containing the **_ _services** section

In addition, the following fields may be included in a service specification:

User Data: *<any arbitrary string>*

The **User Data** field is for the service provider's use and is simply passed along as one of the parameters to the **someMessage:userData:error:** message. This parameter may be useful if multiple services are performed by a single method.

Send Type: *<any valid pasteboard type>*

The **Send Type** field specifies the type of data the requesting application is expected to provide in making the request. You may have more than one **Send Type** field (implying that your request can operate on more than one type of data), but the requesting application is required only to place one of those types into the Pasteboard.

Return Type: *<any valid pasteboard type>*

If the **Return Type** field is specified, then the requesting application will expect you to place some data of that type back into the pasteboard object which you are passed. You may specify any number of return types, but you must place ALL of those types in the pasteboard as part of your implementation of your method (though, of course, you may provide some of them lazilyÐsee the Pasteboard documentation's description of the **provideData:** method). Under normal circumstances, the requestor will use the returned data to replace the selection, though the requestor isn't required to do so.

Executable: *<a full path to an executable file>*

The process which actually services a request need not be a full-fledged application with a user-interface, an icon, and Mach-O segments. The **Executable** field lets you specify the path to the program which should be launched before looking up the port. Note that you must still provide a normal application with a user-interface in whose Mach-O you can put the request information (even if the service is always provided by a lightweight program). This full-fledged application should at the very least give a short description of the provided service(s) as well as any copyright or usage information when the user double-clicks on it from the Workspace.

Timeout: *<some number of milliseconds>*

The **Timeout** field is used to determine how long a request might take to process. The default is 30000 milliseconds. Increasing this time allows time consuming services to be performed before the system assumes there was an error and continues. Decreasing this time for speedy services allows errors to be reported more quickly.

Host: *<the name of a network host>*

The **Host** field lets you specify a specific host on which the service provider should be run. This is done either by requesting the launch of that application from the Workspace Manager running on that host or by using **rsh**(1) to start up the application on the remote host (if it isn't a full-fledged application).

Key Equivalent: *<any character>*

The **Key Equivalent** field may be used to specify the key equivalent for the menu item that invokes the service. Like the Menu Item field, it may be localized by preceding it with a language. For example, a service could have the following entries in its service specification:

```
 Menu Item: Hello
French Menu Item: Bonjour
Key Equivalent: H
French Key Equivalent: B
```

Specifying Services Dynamically

Many services are known in advance, so the services specification is included in a Mach-O section of the executable file. Some services, however, can't be known until run-time. For example, when data is added to a Librarian bookshelf, Librarian can provide a service to look up information within that data.

To facilitate such dynamic services, you must create a text file with a ª.servicesº extension. Alternatively, you may create a directory with a **.services** extension, and a text file called **services** inside it. The format of this text file is exactly the same as the services specification detailed earlier. This file or directory must be placed in your normal application path or one of

/NextLibrary/Services, **/LocalLibrary/Services** or **~/Library/Services**. After adding the file, call the function **NXUpdateDynamicServices()** to get the system to recognize your newly-added services.

Using Services

In order to take advantage of services, an application must have a Services menu, and it must contain Responder objects that register the data types that they may be willing to export and import. If the application's interface is generated with Interface Builder, you can simply drag the Services menu item into the application's menu from an Interface Builder palette. If the application's menu is created programmatically, you can specify the menu item that is to be the Services menu with Application's **setServicesMenu:** method.

Registering Types

Responder objects (including subclasses of View, Window, and Application) should, at the time they are created, register all the data types that they can import and export by using Application's **registerServicesMenuSendTypes:andReturnTypes:** method. The lists of types provided to this method need not be balanced; it's perfectly reasonable for a Responder to handle one export type and three import types, for example. Some of the standard pasteboard data types are listed in **appkit/Pasteboard.h**. A Responder doesn't necessarily have to import or export common data types, but more service providers will be able to act on the common data types than on less common types.

The types supplied to **registerServicesMenuSendTypes:andReturnTypes:** are used to determine which service provider commands are listed in the Services menu. Any service provider that can receive a data type provided by the application or that can supply data to the application should be allowed to have an item in the Services menu, so Responders should provide a complete list of the data they use under any circumstance. The item for an individual service provider will be dynamically enabled any time the application can supply or use the data required or supplied by the service.

The following code could be used to register an object that is, at least in some state, able to export ASCII or RTF text and/or import ASCII text:

```
const char *sendTypes[3];
const char *returnTypes[2];
sendTypes[0] = NXAsciiPboardType;
sendTypes[1] = NXRTFPboardType;
sendTypes[2] = NULL;
returnTypes[0] = NXAsciiPboardType;
returnTypes[1] = NULL;
[NXApp registerServicesMenuSendTypes:sendTypes
    andReturnTypes:returnTypes];
```

Validating Services Dynamically

A Responder (or delegate) that can use services must validate the data types that it can import and export at any given time. It does this by implementing the **validRequestorForSendType:andReturnType:** method. This method is invoked for each service that the application might be able to make use of, with arguments for the data types the service requires. If the Responder can, in its current state, use both the specified send and receive data types (or they are **nil**) it should return **self** to indicate that the corresponding service can be enabled. If the responder can't make use of either the send type or the receive type, it should forward the message to its superclass's implementation; the default implementation will then forward the message up the responder chain, looking for a responder that can take advantage of the service.

The **validRequestorForSendType:andReturnType:** method may be invoked frequently, typically many times per event to ensure that the menu items for all service providers reflect the state of the

application. A Responder's implementation of this method must be fast so that event handling remains snappy. The arguments to this method are NXAtoms, so you can compare the arguments to standard pasteboard types by comparing pointers rather than comparing strings.

The following example demonstrates an implementation of the **validRequestorForSendType:andReturnType:** method for an object that can send and receive ASCII text. Pseudocode is in italics.

```
- validRequestorForSendType:(NXAtom)typeSent
                        andReturnType:(NXAtom)typeReturned
{
    /*
     * First, check to make sure that the types are ones
     * that we can handle.
     */
    if ( (typeSent == NXAsciiPboardType || typeSent == NULL) &&
        (typeReturned == NXAsciiPboardType || typeReturned == NULL) )
    {
        /*
         * If so, return self if we can give the service
         * what it wants and accept what it gives back.
         */
        if ( ((there is a selection) || typeSent == NULL) &&
             ((the text is editable) || typeReturned == NULL) )
        {
            return self;
        }
    }
    /*
     * Otherwise, return the default.
     */
    return [super validRequestorForSendType:typeSent
                                 andReturnType:typeReturned];
}
```

While the application is running, the **validRequestorForSendType:andReturnType:** message is sent to objects in a limited Responder chain, consisting of the responder chain in the key window, the key window's delegate (only if it isn't a Responder), the Application object, and the Application object's delegate (only if it isn't a Responder). The delegates of the key window and Application object are excluded if they are Responders in order to keep the message from being sent down additional responder chains.

How a Service Is Invoked

A service's menu item is enabled any time the application returns a non-**nil** value to a **validRequestorForSendType:andReturnType:** message. If the user then clicks on the service's menu item, the service is invoked. If the service requires data but doesn't send any back (that is, if the service has a send type but no return type) then the service is invoked asynchronously; the application provides the data and continues to run without waiting on the service. However, if the service provides data (that is, the send type is non-NULL) then the service is invoked synchronously; the application won't continue until the service supplies the data or the service request times out.

When the service is invoked, the system checks whether the service requires data. If so, the responder that returned **self** to the **validRequestorForSendType:andReturnType:** message is sent a **writeSelectionToPasteboard:** message to instruct the responder to provide the data it said it would be able to supply. The implementation of this method should put the data on the pasteboard using the **declareTypes:num:owner:** Pasteboard method. If a pasteboard owner is specified, the responder can wait to provide the actual data by implementing the **pasteboard:provideData:** method. (The owner must persist as long as the application is running.) If no owner is specified, the application should provide the data immediately using Pasteboard's **writeType:data:length:** method.

The responder's implementation of **writeSelectionToPasteboard:** should return YES if the selection is successfully written to the Pasteboard, and NO if it fails to supply the data. However, if the responder correctly replies to **validRequestorForSendType:andReturnType:** queries, it should almost always be able to subsequently provide the data.

If the service returns data (that is, has a non-NULL return type), the application will wait (up to the service's time-out period) for the service to provide the returned data. The service must do its processing work and put the data back on the pasteboard using Pasteboard's **declareTypes:num:owner:** method, as described earlier. The application will then receive a **readSelectionFromPasteboard:** message, and its implementation of that method should replace the selection (which could be empty, like a cursor marking an insertion point) with the data from the pasteboard.

Invoking a Service Programmatically

Though services are usually invoked when the user clicks a service menu item, they may also be invoked programmatically with the following function:

```
BOOL NXPerformService(const char *itemName, Pasteboard *pboard)
```

This function returns YES if the service is successfully performed. *itemName* is a Services menu item in any language. Note that Services menu entries which are in subdirectories must include a slash wherever there is a subdirectory, for example, "Mail/Selection". The *pboard* must contain whatever data the service requires, and will, upon return of the function, contain the resultant data provided by the service.

Examples of Services

Here are a few examples of services that have already been implemented to give you an idea of what can be done with NEXTSTEP's services mechanism:

- Optical character recognitionÐWhen a NEXTSTEP application receives a fax, it receives a bitmap that can't be edited as text. If the application is willing to place the image on the pasteboard as a TIFF image, then an optical character recognition service can convert the image to ASCII text and paste it back as editable data.

- EncryptionÐAn encryption service can convert data to a more secure form. For example, Mail can place a mail message on the pasteboard as a standard Rich Text Format (RTF) document, and another application could encrypt the document and place it back into mail as unreadable ASCII text, or as a document to be opened only by another external decryption application.

- Encapsulated PostScript effectsÐMany applications support encapsulated PostScript (EPS) graphic images. An EPS effects service can take selected graphics from the pasteboard, rotate them, scale them, and add other effects before pasting them back. In this manner, consistent graphics editing is enabled, even in applications with minimal graphics support.

- Database lookupÐSelected topics can be looked up in a database. This is a good example of an asynchronous service that reads data from the pasteboard but doesn't send any data back to the main application.

- Document compression and mailingÐOne standard pasteboard type defines a complete file name, including its path. Services use this data to send the current file by Mail, and to compress the current document.

- Macro servicesÐNot all services require data from an application. Some simply provide data on request. Examples include macro programs that insert commonly used data such as signatures and time stamps.