Integrating with External Systems

iPlanet™ Unified Development Server

Version 5.0

August 2001

Contents

List of Figures

List of Tables

List of Procedures

List of Code Examples

Preface

This manual provides reference and usage information about integrating iPlanet UDS applications with external products. It contains instructions for integrating and an appendix with descriptions of sample iPlanet UDS applications demonstrating the concepts found in the manual.

This preface contains the following sections:

Product Name Change

Forte 4GL has been renamed the iPlanet Unified Development Server. You will see full references to this name, as well as the abbreviations iPlanet UDS and UDS.

Audience for This Guide

This manual is intended for application developers. We assume that you:

- have TOOL programming experience
- are familiar with your particular window system
- understand the basic concepts of object-oriented programming as described in *A Guide to the iPlanet UDS Workshops*
- have used the iPlanet UDS workshops to create classes

Organization of This Guide

The following table briefly describes the contents of each part and chapter:

Chapter	Description
Part 1, "Integration with Microsoft Windows Applications"	Provides usage and reference information about integrating iPlanet UDS applications with Microsoft windows applications.
• Chapter 1, "Overview"	Provides an overview about how iPlanet UDS supports Microsoft's OLE 2, ActiveX, and DDE on Windows platforms.
• Chapter 2, "Using OLE to Access Windows Applications"	Explains how you can use OLE linking and embedding and OLE Automation in your iPlanet UDS applications.
• Chapter 3, "Making an iPlanet UDS Service Object an OLE Server"	Describes how to make service objects in an iPlanet UDS application available as OLE servers on the Windows 95 and Windows NT platforms.
• Chapter 4, "Using ActiveX Controls in TOOL Applications"	Explains how you can use ActiveX in the graphical user interfaces of your iPlanet UDS clients that are running in a Windows NT or Windows 95 environment.
• Chapter 5, "Using Dynamic Data Exchange"	Discusses how to enable iPlanet UDS applications to communicate with Windows applications using Dynamic Data Exchange (DDE).

Chapter	Description
Part 2, "Creating an XML Server for UDS Applications"	Provides usage and reference information about exporting iPlanet UDS service objects as XML servers. It also contains information about writing client applications that access the XML server.
• Chapter 6, "Overview of XML Server"	Provides an overview of iPlanet UDS XML servers, including information on how you can create XML servers that can be accessed by external client applications.
• Chapter 7, "Exporting an iPlanet UDS Service Object as an XML Server"	Describes how to export an iPlanet UDS service object as an XML server.
• Chapter 8, "Creating Java Client Applications for an XML Server"	Shows how to create Java client applications that access an iPlanet UDS XML server.
Part 3, "Using External C Functions"	Provides complete information about the C data types you can define in TOOL when you are integrating with external systems. It also provides complete information about integrating with C.
• Chapter 9, "Encapsulating External C Functions"	Discusses how to create classes whose methods are implemented with C functions, store them in a repository as C projects, and use the methods in your TOOL applications.
• Chapter 10, "Making C Functions Available to iPlanet UDS Applications"	Explains how you define a C project whose methods map to C functions.
• Chapter 11, "Writing TOOL Code That Uses C Functions"	Explains how to include C functions in your TOOL application.
• Chapter 12, "TOOL Statements for Defining C Projects"	Contains a reference of the TOOL statements for defining C projects.
• Chapter 13, "Using C Data Types in TOOL"	Explains how iPlanet UDS you can use several standard C data types to pass parameters between iPlanet UDS TOOL methods and certain types of external applications.

Chapter	Description
Part 4, "Writing C++ Client Applications"	Provides complete information about integrating with C++.
• Chapter 14, "Accessing iPlanet UDS Using C++"	Explains how you can generate a C++ API that lets you access your iPlanet UDS application using C++ calls.
• Chapter 15, "C++ API Reference Information"	Describes the handle classes that are generated by iPlanet UDS when you have iPlanet UDS generate a C++ API for a client partition.
Part 5, "Using Network and Operating System Features"	Describes how you can use system activities and network sockets to enable your application to communicate with an iPlanet UDS applications.
• Chapter 16, "Using System Activities and Network Connections"	Discusses how to interact with other applications using system activities and network connections.
Appendix A, "iPlanet UDS Example Applications"	Provides instructions for using the example applications used in this manual.
Appendix B, "Olegen Mapping Conventions"	Describes how the `Olegen` utility interprets the interfaces provided by OLE servers and ActiveX controls.

Text Conventions

This section provides information about the conventions used in this document.

Format	Description
italics	Italicized text is used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.
ALL CAPS	Text in all capitals represents environment variables (FORTE_ROOT) or acronyms (UDS, JSP, iMQ). Uppercase text can also represent a constant. Type uppercase text exactly as shown.
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Other Documentation Resources

In addition to this guide, iPlanet UDS provides additional documentation resources, which are listed in the following sections. The documentation for all iPlanet UDS products (including Express, WebEnterprise, and WebEnterprise Designer) can be found on the iPlanet UDS Documentation CD. Be sure to read "Viewing and Searching PDF Files" on page 32 to learn how to view and search the documentation on the iPlanet UDS Documentation CD.

iPlanet UDS documentation can also be found online at http://docs.iplanet.com/docs/manuals/uds.html.

The titles of the iPlanet UDS documentation are listed in the following sections.

iPlanet UDS Documentation

- *A Guide to the iPlanet UDS Workshops*
- *Accessing Databases*
- *Building International Applications*
- *Escript and System Agent Reference Guide*
- *Fscript Reference Guide*
- *Getting Started With iPlanet UDS*
- *Integrating with External Systems*
- *iPlanet UDS Java Interoperability Guide*
- *iPlanet UDS Programming Guide*
- *iPlanet UDS System Installation Guide*
- *iPlanet UDS System Management Guide*
- *Programming with System Agents*
- *TOOL Reference Guide*
- *Using iPlanet UDS for OS/390*

Express Documentation

- *A Guide to Express*
- *Customizing Express Applications*
- *Express Installation Guide*

WebEnterprise and WebEnterprise Designer Documentation

- *A Guide to WebEnterprise*
- *Customizing WebEnterprise Designer Applications*
- *Getting Started with WebEnterprise Designer*
- *WebEnterprise Installation Guide*

Online Help

When you are using an iPlanet UDS development application, press the F1 key or use the Help menu to display online help. The help files are also available at the following location in your iPlanet UDS distribution: `FORTE_ROOT/userapp/forte/cl`*n*`/*.hlp`.

When you are using a script utility, such as Fscript or Escript, type help from the script shell for a description of all commands, or help <*command*> for help on a specific command.

iPlanet UDS Example Programs

A set of example programs is shipped with the iPlanet UDS product. The examples are located in subdirectories under `$FORTE_ROOT/install/examples`. The files containing the examples have a `.pex` suffix. You can search for TOOL commands or anything of special interest with operating system commands. The `.pex` files are text files, so it is safe to edit them, though you should only change private copies of the files.

Viewing and Searching PDF Files

You can view and search iPlanet UDS documentation PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

NOTE You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

➤ **To copy the documentation to a client or server**

1. Copy the `doc` directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the `doc` directory; the location is not dependent on the iPlanet UDS distribution.

2. Set up a directory structure that keeps the `udsdoc.pdf` and the `uds` directory in the same relative location.

   The directory structure must be preserved to use the Acrobat search feature.

NOTE To uninstall the documentation, delete the `doc` directory.

➤ **To view and search the documentation**

1. Open the file `udsdoc.pdf`, located in the `doc` directory.
2. Click the Search button at the bottom of the page or select Edit > Search > Query.
3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click Search.

   A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

   NOTE For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

   All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	Keyboard Command
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

   To return to the `udsdoc.pdf` file, click the Homepage bookmark at the top of the bookmarks list.

6. To revisit the query results, click the Results button at the bottom of the `udsdoc.pdf` home page or select Edit > Search > Results.

Part 1

Integration with Microsoft Windows Applications

Part 1 of *Integrating with External Systems* provides usage and reference information about integrating iPlanet UDS applications with Microsoft windows applications.

Part 1 contains the following chapters:

Chapter 1

Overview

iPlanet UDS provides ways for you to use Microsoft's OLE Version 2 (OLE 2) methods in your iPlanet UDS applications, and ways for OLE clients to access service objects in iPlanet UDS applications. You can also incorporate ActiveX controls into your iPlanet UDS applications.

The chapters in this part discuss the following topics:

- embedding and linking external OLE-enabled Windows objects in an iPlanet UDS window using an OLE field
- interacting with Windows applications within your iPlanet UDS application
- making iPlanet UDS service objects available as OLE servers
- using ActiveX custom controls in iPlanet UDS applications
- using DDE interfaces to interact with Window applications

This chapter provides an overview of these features.

About OLE, ActiveX, and DDE

This section provides an overview about how iPlanet UDS supports Microsoft's OLE 2, ActiveX, and DDE on Windows platforms.

About OLE

To use OLE 2, you need to have an OLE server application installed. OLE servers are expected to provide the OLE shared libraries you need to use either an OLEField or the classes provided by the iPlanet UDS system OLE library.

OLE is a mechanism for interacting with objects associated with Windows applications. iPlanet UDS can interact with two main parts of OLE: object linking and embedding and OLE Automation. The explanations in this chapter assume that you understand the concepts of OLE and have access to information about using OLE, including object linking and embedding and OLE Automation.

OLE is based on a set of interfaces provided by many Windows applications. A Windows program can interact with the objects associated with another Windows application. These two programs are known, respectively, as the *client* and the *server*. An OLE server is a program that has access to data and that provides functions that might be useful to other programs. An OLE client is a program that obtains this data or interacts with objects associated with the server. A client corresponds to an OLE controller.

An *OLE object* is anything that can be considered a "thing" in Windows. For example, applications, documents, and interfaces can all be objects to OLE.

Object Linking and Embedding

iPlanet UDS provides a class in the Display Library called OLEField. This OLEField class supports most object linking and embedding functions provided by Windows applications. For example, in the Window Workshop, you can create a new OLE field, then embed part of a Microsoft Word for Windows document into your window. For more information about using OLE fields, see "Using Object Linking and Embedding" on page 42.

Using Windows Applications

When iPlanet UDS invokes methods on OLE server applications, iPlanet UDS is acting as an *OLE automation controller*, which means that iPlanet UDS applications can use interfaces provided by other Windows applications. For more information about invoking methods on Windows applications, see Chapter 2, "Using OLE to Access Windows Applications."

iPlanet UDS provides a library called OLE that lets you invoke functions calls on external Windows programs from within your TOOL methods. You can either use an iPlanet UDS utility, `Olegen`, to generate a TOOL project that contains methods that map to the functions provided by an OLE Automation interface, or you can invoke a function directly using methods provided by iPlanet UDS.

Defining Service Objects as OLE Servers

When an iPlanet UDS service object provides an OLE interface to OLE client applications, iPlanet UDS is acting as an OLE automation server, or *OLE server*. For information about how to define an iPlanet UDS service object as an OLE server, see Chapter 3, "Making an iPlanet UDS Service Object an OLE Server."

ActiveX Controls

You can also add *ActiveX controls*, which are sometimes called OLE custom controls and OCX controls, to your iPlanet UDS graphical user interfaces. iPlanet UDS provides a Display library class named `ActiveXField` that lets you add an ActiveX control to an iPlanet UDS window in the Window Workshop.

For information about using ActiveX controls, see Chapter 4, "Using ActiveX Controls in TOOL Applications."

Terminology Used in This Part

Because this documentation integrates two independent systems, there may be some confusion about our terminology. The following list defines terms that are specific to integrating with external Windows applications using OLE:

ActiveX control (also called OCX control or OLE custom control) a specialized OLE server that provides small, self-contained functions with a graphical interface.

client (OLE) an OLE-enabled application (object) that requires the services of another Windows application (server).

DDE (Dynamic Data Exchange) a mechanism for interprocess communication supported in Windows applications.

embedding (OLE) inserting an object in your client application. This object might be editable by another OLE-enabled server application.

interface (OLE) an object that provides the means of invoking a group of related functions belonging to an object.

linking (OLE) establishing a connection between the client application and an object that uses the server application.

object (OLE) anything that can be considered a "thing" in Windows. For example, applications, documents, and interfaces can all be objects to OLE.

object linking and embedding defines how part of one object created using one application is associated with another object created using another application. For example, part of a Microsoft Excel spreadsheet can appear as part of a Microsoft Word for Windows document.

OLE 2 Microsoft's specification for how Windows objects interact.

OLE Automation a set of interfaces that enables an application to be used as an OLE object.

OLE automation controller an application that uses external Windows objects using OLE Automation interfaces.

OLE server an application that supplies OLE Automation interfaces that can be accessed by a client application.

OLE library (iPlanet UDS) a library provided by iPlanet UDS, which supports the functions needed to integrate with OLE.

server (OLE) an OLE-enabled application (object) that provides services to another Windows application (client).

Chapter 2

Using OLE to Access Windows Applications

iPlanet UDS supports two features that lets you use OLE to access the functions provided by Windows applications:

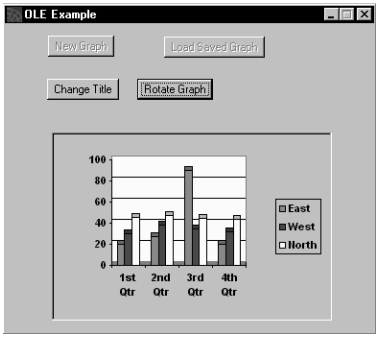
- OLE linking and embedding using the OLEField class
- OLE Automation, which lets you invoke TOOL methods that invoke methods on a Windows application

This chapter explains how you can use OLE linking and embedding and OLE Automation in your iPlanet UDS applications.

About Using OLE to Access Windows Applications

In an iPlanet UDS application, you can use *OLE linking and embedding* to display a Windows document in your iPlanet UDS application. The end user can then interact with the document using the Windows application that created it. For example, you can include a portion of an Microsoft Graph chart in a TOOL window, and an end user can double-click the chart to edit it using Microsoft Graph.

Figure 2-1 A Microsoft Graph Chart in a TOOL window



"Using Object Linking and Embedding" explains how you can use the `OLEField` class to include Windows documents in your applications.

OLE Automation You can also write TOOL code that interacts with Windows applications using OLE Automation. OLE Automation defines a set of common interfaces that Windows applications (*OLE servers*) can provide. Other Windows applications (*OLE clients*) can use these common interfaces to invoke the OLE methods supported by the OLE server.

"Using OLE Automation" on page 50 begins a description of how to write TOOL code that interacts with a Windows application.

In addition to OLE linking and embedding and OLE Automation, you can also embed ActiveX controls. For information about using ActiveX controls in your TOOL application, see Chapter 4, "Using ActiveX Controls in TOOL Applications."

Using Object Linking and Embedding

The simplest type of integration between iPlanet UDS and a Windows applications is object linking and embedding. This type of integration lets you either:

- link an iPlanet UDS OLE field to all or part of a Windows document managed by a Windows application

  In this case, any changes made by the iPlanet UDS application are maintained in the original Windows document.

- embed all or part of a Windows document in an OLE field and its associated cache file

  In this case, any changes made by the iPlanet UDS application must be maintained in a cache file, and the original Windows document does not change.

  A *cache file* is a file that stores information about the linked or embedded object. If you do not define an embedded OLE object as a cached object with a cache file, then the user of your application cannot save changes made to the embedded OLE object. Similarly, you cannot save information about how a linked object appears in the iPlanet UDS application unless the linked object is a cached object.

When the user runs the iPlanet UDS application containing OLE fields, she can change the linked or embedded object by double-clicking the OLE field in the iPlanet UDS window to start the Windows application that manages the document. This type of editing is called *in-place activation* or editing in place.

Your TOOL code can interact with the managing application for the linked or embedded object if you use OLE automation methods, as described in Chapter 2, "Using OLE to Access Windows Applications."

There are two ways to include OLE fields in a TOOL application: defining an OLEField widget in the Window Workshops, or instantiating and defining an OLEField object in TOOL.

Defining an OLE Field in the Window Workshop

In general, to define an OLE field, you need to create an OLE field and define the OLE object that is linked or embedded in the OLE field.

This section describes the simplest, most common ways to define an OLE linked or embedded object in the Window Workshop.

NOTE You can define OLE fields on any platform. However, you can only insert OLE objects and run applications that use OLE objects if you are running iPlanet UDS on Windows machines that have the required OLE server applications installed.

Creating an OLE Field

These instructions describe how to create a new OLE field in the Window Workshop. The following sections describe how to define the kind of OLE object in the OLE field.

➤ **To create an OLE field**

1. Choose the OLE button on the tool bar. Draw the OLE field in the window.
2. Double-click the OLE field to open its property dialog.

3. Specify the name of the OLE field in the Attribute Name field.

If you plan to write TOOL methods that invoke OLE Automation methods, you should set the Mapped Type field to the appropriate subclass of CDispatch. For more information about possible subclasses of CDispatch, see "Generating TOOL Projects That Access OLE Methods" on page 51. The data type for the mapped attribute is CDispatch, by default, or the specified subclass of CDispatch. If you delete CDispatch from the Mapped Type field, the OLEField widget will not have a mapped attribute.

OLEField Properties Dialog

The OLEField Properties dialog has the following fields and buttons:

Use This Property	For This Purpose
Attribute Name	Sets an attribute name for the picture field.
Mapped Type	Specifies the mapped data type for the OLE field. This value must be CDispatch or a subclass of CDispatch.
Allow Activate in Place	Sets whether to start the application for the current object as part of the current window.
Allow In Place Toolbar	Sets whether to display the tool bar for the application activated in-place as part of the current window.

Use This Property	For This Purpose
Insert Cached Object button	Allows you to add a linked or embedded object that can be saved. The dialog that this button displays is provided by Windows, so you must be running on a Windows machine to access it. See your Windows documentation for information about the Insert Cached Object dialog.
Insert Object button	Lets you add a linked or embedded object. This button displays a dialog, in which you can define the linked or embedded OLE object. To add a new file, following the instructions in "To define a new OLE object" on page 47. To insert an object based on an existing object, see "To create an embedded object based on an existing file" on page 48.
Load Cache File button	Lets you link to an OLE object stored in a cache file. This button displays a file selection dialog to select the cache file.
Help Text	Opens the Help Text dialog for the field.
Size Policy	Opens the SIze Policy dialog for the field.

Figure 2-2 OLEField Properties Dialog

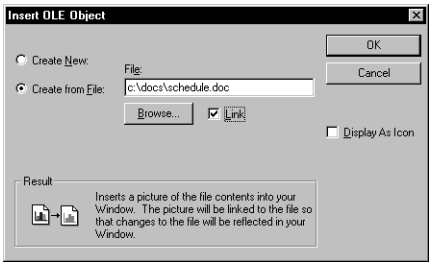
Cache File Name is a read-only field that contains the name and path of the cache file, which contains saved information about the linked or embedded OLE object.

Linking to an OLE Object

These instructions describe how to link to an OLE object. When you run an application containing a linked object, the end user can (if the OLEField is in Update mode), start the application that manages the object and edit the object.

➤ **To link to an OLE object**

1. In the OLEField Properties dialog, choose the Insert Object button to open the Insert OLE Object dialog.



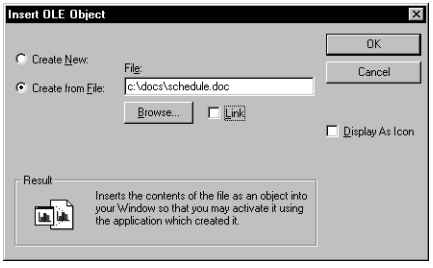
2. Choose Create from File.
3. Select a file path and name.
4. Click the Link check box.
5. Click OK.

Embedding a Read-Only OLE Object

These instructions describe how to embed an OLE object. When you run an application containing this embedded OLE object, the end user can only look at the object.

➤ **To create a read-only embedded object based on an existing file**

1. Choose the Insert Object button to open the Insert OLE Object dialog.



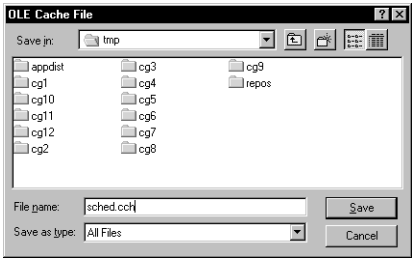
2. Choose Create from File.
3. Select a file path and name.
4. Click OK.
5. Set the initial state of the OLE field to View, Disable, or Inactive.

Embedding an Editable OLE Object

These instructions describe how to embed an editable OLE object. When you run an application containing this embedded OLE object, the end user can double-click on the OLE field to start the managing Windows application. Any changes that the user makes to the OLE object are saved when she closes the TOOL window containing this OLE field.

➤ **To create an editable embedded object**

1. Choose the Insert Cached Object button to open the OLE Cache File dialog.



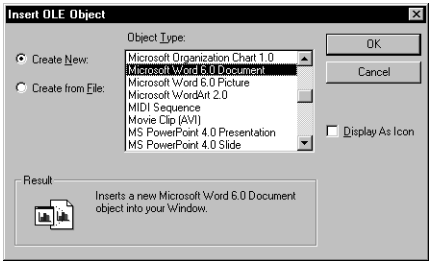
2. Specify the name of a new file in which to save information about the embedded OLE object. Click Save.

   The Insert OLE Object dialog opens.

3. Follow either of the following sets of instructions to define a new OLE object or create an OLE object from a file.

➤ **To define a new OLE object**

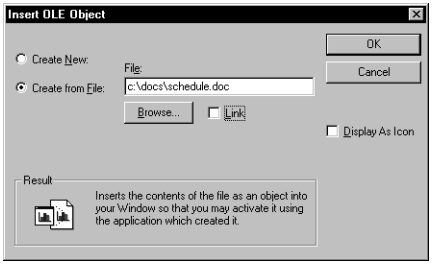
1. Choose the Create New radio button and an object type in the Object Type field.



2. Click OK.

➤ **To create an embedded object based on an existing file**

1. Choose Create from File.



2. Select a file path and name.
3. Click OK.

Defining an OLE Field in TOOL

This section describes how to define an OLEField object in TOOL.

➤ **To define an OLEField object in TOOL**

1. Define and instantiate an OLEField object:

```
myOLEField : OLEField = new;
```

2. Use an OLEField method to create an embedded or linked object. The following table lists the various options for creating an embedded or linked object with the appropriate OLEField method:

Contents of the OLE Field	OLEField Method
User-specified linked or embedded object	InsertOLEObject
New embedded object	CreateEmbeddedObjectFromProgID CreateEmbeddedObjectFromCLSID
Embedded object based on an existing file	CreateEmbeddedObjectFromFile
Linked object based on an existing file	LinkTo
Linked object based on a cache file	LoadObjectFromCacheFile ReadFromFile
Linked object based on the document in the clipboard	PasteLink

3. Specify the Parent attribute of the OLEField widget as the window or the grid where you want the OLE field to appear.

OLE Menu Groups

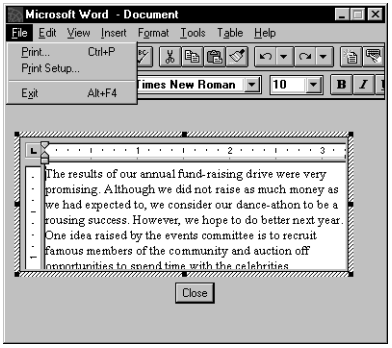
When you start the Windows application in place to edit an embedded object, iPlanet UDS merges the menus for the TOOL application and the Windows application based on the settings defined for OLE menu groups. *OLE menu groups* represent the three groups that Microsoft defines for submenus: File, View, and Window. You can specify that a TOOL application submenu belong to one of these groups or be invisible.

When you define the TOOL application that contains the OLE field, you can define the OLE menu groups in the Menu Workshop.

➤ **To define OLE menu groups in the Menu Workshop**

1. Select a submenu of the TOOL application window.
2. Choose Item > OLE Menu Group, then one of the following submenu list items:
   - Invisible, the default, does not display the submenu when the Windows application is activated.
   - File Group displays the submenu before the File menu for the Windows application.
   - View Group displays the submenu before the View menu for the Windows application.
   - Window Group displays the submenu before the Window menu for the Windows application.

The following figure shows how the menus might look when Microsoft Word is activated in place. The File menu is a TOOL menu, which has an OLE Menu Group setting of File Group. This TOOL application also has an Edit menu and a Help window, both of which have OLE Menu Group settings of Invisible, so these menus are not merged into the menus for Microsoft Word.



Using OLE Automation

iPlanet UDS fits the category of an *OLE automation controller*, which means that iPlanet UDS applications can use interfaces provided by Windows applications that are OLE servers.

iPlanet UDS provides a library called OLE that lets you invoke functions on Windows applications that are OLE servers from within your TOOL methods. This manual refers to these function calls as *OLE methods*. You can invoke OLE methods using either of the following ways:

- Use an iPlanet UDS utility, `Olegen`, to generate a TOOL project that contains methods that map to OLE methods provided by an OLE Automation interface. This approach is explained in "Generating TOOL Projects That Access OLE Methods" on page 51.

  One advantage of using `Olegen` to generate a TOOL project is that you can generate a static interface to the OLE server application. When you import this project, iPlanet UDS is aware of the syntax for a particular method call, so the iPlanet UDS compiler can check for syntax errors.

Another advantage of this approach is that the `Olegen` utility uses the names of functions and parameters provided by the OLE server to generate classes, methods, and method parameters in the generated TOOL project. Having this information available within your TOOL development environment can make it easier to locate methods and define parameters for the OLE functions you want to use.

- Invoke an OLE method directly using methods provided by iPlanet UDS. This approach is explained in "Invoking Methods on OLE Interfaces Using CDispatch" on page 60.

  The main advantage of this approach is that you do not need to run the `Olegen` utility to create a static interface, nor do you need to import the resulting TOOL project. Therefore, this approach can save time and resources, particularly if the OLE server has several extensive interfaces.

For reference information about the classes in the iPlanet UDS OLE library, see the iPlanet UDS online Help.

iPlanet UDS provides an example called OLESample, which demonstrates how you can use an OLEField with OLE Automation methods. For information about locating and using this example, see "OLESample" on page 353.

Generating TOOL Projects That Access OLE Methods

This section briefly describes the steps you need to perform to implement an iPlanet UDS client application that interacts with Windows server applications.

iPlanet UDS provides a set of classes that allow iPlanet UDS applications to interact with Windows applications using OLE Automation interfaces. An iPlanet UDS application can therefore be an OLE client. For example, an iPlanet UDS inventory application (client) might want to allow the user to update a certain Excel spreadsheet (server). Using OLE methods in TOOL, you can write an iPlanet UDS application that starts Excel and opens a spreadsheet.

➤ **To write an iPlanet UDS application that uses TOOL methods generated by Olegen**

1. Generate TOOL classes for the OLE application using the `Olegen` utility, then import the .pex file into your development repository.
2. Write the iPlanet UDS application that uses the OLE methods.

3. Partition the iPlanet UDS application components that use OLE methods on the appropriate Windows nodes and make a distribution for the iPlanet UDS application.
4. Install the iPlanet UDS application.

These steps are described in greater detail in the following sections.

Generate TOOL Classes for the OLE Application

iPlanet UDS provides an `Olegen` utility that generates a TOOL project definition using the OLE Automation interface information provided by the specified Windows program. Only Windows programs that are OLE servers provide this interface information.

Before you can interact with an OLE server application using TOOL methods corresponding to the OLE application's interface, you need to have TOOL classes that correspond to that application in your development repository. If these TOOL classes are already available in your development repository, you can skip this section and move on to "Write the iPlanet UDS Application Using OLE Methods" on page 54.

Running the Olegen Utility

To run the `Olegen` utility, you must be running on Windows. You can start this utility in the Windows dialog that you can access by selecting the `Run` command from the File menu in Program Manager.

The syntax for starting the `Olegen` utility is:

olegen *input_specification* [*output_specifications. . .*] **[-ai]**

input_specification is one of the following:

input_specification	Description
-it *type_library*	*type_library* is the file name of the type library for the Windows program, if available.
-ip *ProgID*	*ProgID* is the programmatic identifier of an OLE server class. These identifiers typically have the syntax *application_name.object*, for example, "excel.application."
-ic *CLSID*	*CLSID* is the unique identifier string for an OLE server class. The CLSID is a string of 32 hex digits enclosed in braces, for example: {00021A00-0000-0000-C000-000000000135}.

output_specifications, which are optional, are one or more of the following:

output_specifications	Description
-of *output_file_name*	Specifies the file name for the generated .pex file.The default file name is `olegen.pex`.
-op *output_project_name*	Specifies the name of the generated project. Ignored if type libraries are available. If `Olegen` can access a type library, then the project name is the type library name. If `Olegen` *cannot* access a type library, then the default project name is UnknownProject.
-oc *output_class_name*	Specifies the name of the generated class. Ignored if type libraries are available. If `Olegen` can access a type library, the class names are one or more dispatch interface names. If `Olegen` *cannot* access a type library, the default class name is UnknownInterface.

The **-ai** flag ("assume input"), which is optional, specifies how the `Olegen` utility interprets method parameters that do not specify a passing mode. By default, `Olegen` interprets all method parameters that do not specify a passing mode as input Variant objects. When the **-ai** flag is specified, `Olegen` interprets all parameters that do not specify a passing mode as input parameters of an iPlanet UDS data type.

NOTE Because it is usually easier to work with iPlanet UDS data types than with Variant objects, we recommend that you specify the **-ai** flag. The default is not to assume that parameters are input parameters to be compatible with earlier releases of iPlanet UDS.

The `Olegen` utility automatically maps the OLE Automation interface provided by the specified Windows program to TOOL method syntax, using whatever information the Windows program provides. To generate a project definition that maps to the Microsoft Graph application, you can start the `Olegen` utility using the following command:

```
olegen -it c:\windows\msapps\msgraph5\gren50.olb
    -of c:\examples\extsys\ole\msgraph.pex
```

For information about the type library name, the programmatic ID, or the CLSID for a given Windows application, see the documentation for that application.

For information about how `Olegen` interprets the information in a type library, see Appendix B, "Olegen Mapping Conventions."

Importing the Generated Project Definition .pex File

To import the .pex file generated by the `Olegen` utility, use either the `Import` command in the Repository Workshop, or the `ImportPlan` command in Fscript. For specific instructions for importing a project definition, see *A Guide to the iPlanet UDS Workshops* or the *Fscript Reference Guide*.

Write the iPlanet UDS Application Using OLE Methods

Once you have imported the project definition, you can use the methods in this project almost as though they were native TOOL methods.

If you are planning to invoke these methods on an OLE object that has been linked or embedded in an OLE field, you should consider defining the mapped type of the OLE field as the generated CDispatch subclass for the OLE object. For more information about using OLE fields, see "Using Object Linking and Embedding" on page 42.

To write TOOL code that interacts with a Windows server program, for example, Microsoft Graph, you perform the following steps:

1. Include the OLE library and the generated project that you just imported as suppliers to the main TOOL project.

```
includes OLE;
includes Graph; -- The project that contains the OLE dispatch
--interface class and OLE methods for Microsoft Graph.
includes DisplayProject;
includes Framework;
```

Project: OLESample

2. Within the TOOL code, declare and instantiate the object containing the OLE method you want to use. This class maps to a particular dispatch interface provided for a Windows application, and is a subclass of CDispatch.

```
graphChart : Graph.Chart = new;
```

3. Within the TOOL code, set the ObjectReference attribute of the object. You need to set the ObjectReference attribute of the dispatch interface before you can call any methods. The ObjectReference attribute is a pointer that references a dispatch interface (IDispatch) on an OLE object.

   You can set the value of ObjectReference using the OLEObjectReference attribute in an OLE field, a moniker, a ProgID, a CLSID, or the ObjectReference value of another CDispatch object. For a detailed description of how to set the ObjectReference attribute, see the iPlanet UDS online Help.

   The following example shows how you can set the ObjectReference for the graphChart object using the ObjectReference attribute of the <OurChart> OLE field:

```
graphChart.ObjectReference = <OurChart>.OleObjectReference
```

4. Within the TOOL code, invoke methods on the object referenced by the ObjectReference attribute. If the methods include parameters with data type `Variant`, you need to check the OLE server documentation to ensure that you are passing data of the correct data type for the OLE method. For more information about these parameters, see "Mapping Data Types in TOOL" on page 363.

   The following example shows how you can define a title on a Microsoft Graph graph:

```
graphChartTitle : Graph.ChartTitle = new;
graphChartTitle.SetDispatchObject(graphChart.ChartTitle);
graphChartTitle.Text = VariantString(value = userTitle);
```

   Project: OLESample • **Class:** OLEWindowClass • **Method:** Display

   To pass data to partitions not on a Windows platform, you need to copy data from a restricted OLE object, such as VariantString, to an object of a non-restricted class, such as TextData.

For information about handling exceptions raised when you are working with OLE methods, see "Handle Any Exceptions" on page 65.

Dealing with Variant Objects

If the methods include parameters with data type `Variant`, you need to check the OLE server documentation to ensure that you are passing data of the correct data type for the OLE method.

When you are using a method or attribute that uses a data type of `Variant`, `VariantString`, or other subclasses of `Variant`, you need to:

- convert your TOOL objects and data into an object of the `Variant` class or subclass
- convert the object of the `Variant` class or subclass into a TOOL object or data type

These conversions are discussed in the following sections. For information about the `Variant` class and its subclasses, see the iPlanet UDS online help.

Converting Data to a Variant Object

When you pass data to the OLE server using a generated TOOL method or attribute, you frequently need to pass an object of the `Variant` class or one of Variant's subclasses.

You can avoid using Variant objects by using the **-ai** flag of the `olegen` command when you generate the TOOL classes. For more information, see "Generate TOOL Classes for the OLE Application" on page 52.

In most cases, you will interact with an OLE server using the generated attributes. Many of these attributes are defined to have Variant objects as attributes.

First, you need to check the documented interface for the OLE server to determine what data type is expected by the OLE server for that attribute.

For example, a property of the SSTab Dialog control is called `TabOrientation`. The TOOL class, `SSTab`, that corresponds to that control has an attribute that is also called TabOrientation. This attribute's value is an object of the `Variant` class. According to the documentation available for the SSTab Dialog control, the data expected for this property is an integer value that can be represented by a constant.

To specify an integer value for this attribute, you can write code like the following:

```
currTabDialog : SSTab = new;
-- Define a VariantInteger object and assign it the value for the
-- TabOrientation property.
tabOrientValue : VariantInteger = new;
tabOrientValue.Value = 1;
-- Assign the property value to the corresponding attribute.
currTabDialog.TabOrientation = tabOrientValue;
```

Another common case that occurs when you work with an OLE server application is that you are dealing with objects contained by other objects. For example, in Microsoft Excel, a Workbook contains a Worksheet, which contains a Sheet, which contains a ChartObject, which contains a Chart. Therefore, you need to navigate through this hierarchy to access a Chart object. Usually container objects have properties or methods that let you access objects that they contain.

However, in many cases, the methods or attributes that provide access to objects in OLE server applications provide the objects as Variant objects. Before you can access the methods or properties of the container objects to get to the contained objects, you need to get a handle to the CDispatch object (the dispatch interface) for the container object.

➤ **To access objects that are contained in a container object defined as a Variant object**

1. Use the `CDispatch.SetDispatchObject` method with the container object to get a handle to the dispatch interface for that object.
2. Use the methods and attributes defined for the container object to access its contained objects.

```
-- This method copies the specified chart to the clipboard.
-- An Excel Worksheet is already open.
thisChartObject : Excel.ChartObject = new;
thisChart : Excel.Chart = new;
wshtVar : Variant;
currWorksheet : Excel.Worksheet = new;
-- Get a handle to the active worksheet.
wshtVar = self.thisExcel.ActiveSheet;
thisChartObject : Excel.ChartObject = new;
-- Get a handle to the dispatch interface for the worksheet.
currWorksheet.SetDispatchObject(variantRef = wshtVar);
chartNameVar.Value = chartName.Value;
chartNameVar : VariantString = new;
-- Get a handle to the chart object.
chartObjVariant : Variant = currWorksheet.ChartObjects(
  index=chartNameVar);
-- Get a handle to the dispatch interface for the chart object.
thisChartObject.SetDispatchObject(variantRef = chartObjVariant);
chartVariant : Variant = thisChartObject.Chart;
thisChart.SetDispatchObject(variantRef = chartVariant);
thisChartObject."Copy"();
```

Converting a Variant Object to a TOOL Object or Data Type

If you retrieve data from a property or get a return value from a method that is an object of class `Variant` or one of its subclasses, you need to convert the value to a TOOL object or data type before you can use the data in your TOOL code.

For example, if a method returns a `Variant` object, but you know from the documentation that the parameter of the method actually returns a string, you need to use the `VariantString` class to get the value, then convert the data to a TOOL object.

You can avoid using Variant objects by using the **-ai** flag of the `olegen` command when you generate the TOOL classes. For more information, see "Generate TOOL Classes for the OLE Application" on page 52.

The following example shows how you can convert a value retrieved from the cell of an Excel spreadsheet to a TextData object. In this example, the `Range` class is defined by the project definition generated by `Olegen` for Excel.

```
-- Each cell is identified using its row and column.
-- Get a reference to a range.
currRangeVariant : Variant = new;
currRangeVariant = self.ThisExcel.Cells(rowIndex = CellRow,
  columnIndex = CellColumn);
-- Get a handle to the dispatch interface for the range.
currRange : Range = new;
currRange.SetDispatchObject(variantRef=currRangeVariant);
-- Assign the value of the cell in the range to a TextData.
-- (Note the cast of the range value to a VariantDouble to get
-- the double data.)
retData : TextData = new;
retData.DoubleValue = (VariantDouble(currRange.Value).Value);
```

To pass data to partitions not on a Windows platform, you need to copy data from a restricted OLE object, such as VariantString, to an object of a non-restricted class, such as TextData.

Partition the TOOL Application and Make a Distribution

TOOL classes that subclass or instantiate OLE Automation interface classes can only run in client partitions that have Windows installed. Start the Partition Workshop to examine the default configuration for your application, and remove these client partitions from nodes where you do not want these partitions installed.

You can make the distribution just as you would for any regular TOOL application. You can either make the distribution from the Partition Workshop or from within Fscript. For more information about using the Partition Workshop or Fscript, see *A Guide to the iPlanet UDS Workshops* or the *Fscript Reference Guide*.

Install the Client Application

You can install the client application just as you would any other TOOL client application. For information about installing iPlanet UDS applications, see *iPlanet UDS System Management Guide*.

Invoking Methods on OLE Interfaces Using CDispatch

This section discusses how you can use CDispatch to invoke methods on OLE server applications.

iPlanet UDS provides a `CDispatch` class as part of the OLE library. The `CDispatch` class contains a pointer to the dispatch interface (IDispatch) for a particular OLE server object. An OLE server object can be an application, such as Microsoft Excel, or an object associated with the application, for example, a Range or Charts object in Microsoft Excel.

The methods of the `CDispatch` class let you:

- set the CDispatch ObjectReference attribute to reference a particular OLE server object
- invoke methods provided by that OLE server

Therefore, you can use the `CDispatch` class to invoke methods on OLE server interfaces without generating methods for all the available OLE Automation interfaces. The `CDispatch` class is fully described in the iPlanet UDS online Help.

If you want to generate TOOL classes for an OLE server, see "Generating TOOL Projects That Access OLE Methods" on page 51.

To write TOOL code that invokes methods on the OLE server, perform the following steps:

1. Decide which OLE methods you want to use.
2. Include the OLE library as a supplier plan to the current TOOL project.
3. Instantiate an object of the `CDispatch` class from the OLE library using the keyword `new`.
4. Set the ObjectReference attribute of the CDispatch object.
5. Set the parameters you need to pass to the OLE method.
6. Use the `InvokeMethod` or `InvokeMethodWithResult` method on the `CDispatch` class to invoke the OLE method.

7. Check the results of the method.
8. Handle any exceptions.
9. Partition the client application on the appropriate Windows nodes. and make a distribution.
10. Install the client application.

There are no special steps for deploying an application that uses OLE Automation methods. However, remember that a TOOL component that uses an OLE Automation method can only be partitioned on a node running Windows.

To pass data to partitions not on a Windows platform, you need to copy data from a restricted OLE object, such as VariantString, to an object of a non-restricted class, such as TextData.

Decide Which OLE Methods to Invoke

Windows applications that can be OLE servers provide ways for a client application to learn about the OLE Automation interfaces and methods that are available.

Usually, the application also documents these interfaces and methods, either in the online help or the documentation provided with the product. For example, Microsoft Excel documents its OLE Automation interfaces as its means of interfacing with Microsoft Visual Basic, while Microsoft Word for Windows documents many of its available OLE Automation interfaces as Word Basic commands.

You need to refer to the documentation provided for the Windows application to learn the following:

- what OLE object the method belongs to
- the method name
- the parameters for the method, their purposes, and their data types

Include the OLE Library as a Supplier Plan

Your TOOL project contains the following lines to indicate that it includes supplier plans, which are Framework and the OLE library:

```
includes Framework;
includes OLE;
```

You can also include your OLE library as a supplier plan from within the Project Workshop. For more information, see *A Guide to the iPlanet UDS Workshops*.

Instantiate an Object of the CDispatch Class

Within your TOOL code, you must instantiate an object of the `CDispatch` class, as shown in the following example:

```
wordApp : CDispatch = new;
```

Set the ObjectReference Attribute

You need to set the ObjectReference attribute of the CDispatch object before you can call any methods. The ObjectReference attribute is a pointer that references a dispatch interface (IDispatch) on an OLE object. You can set the value of ObjectReference using:

- the OLEObjectReference attribute in an OLE field
- a moniker, a ProgID, or a CLSID
- the ObjectReference value of another CDispatch object

For more information about setting the ObjectReference attribute, see the iPlanet UDS online Help.

Set the Parameters You Need

Before you can invoke a method that requires parameters, you need to create an array of parameters that iPlanet UDS will pass to the OLE method. You can either specify parameters with names, or you can specify parameters by position.

To specify parameters with names, you can use an Array of NamedParameter. To specify parameters by position, you can use Array of Variant. TOOL passes positional parameters according to their left-to-right order in the method definitions.

The following example shows how you declare and define a list of named parameters:

```
namedParams : array of NamedParameter = new;
namedParams.appendRow(
  NamedParameter(Name='Name',
    Value=VariantString(value='myfile.doc'));
```

In this example, the Microsoft Word for Windows method FileOpen requires only one parameter, so the namedParams Array of NamedParams contains only one object.

The following example shows how you declare and define a list of positional parameters:

```
unnamedParams : Array of Variant= new;
unnamedParams.appendRow(VariantString(value='R')); // Find
unnamedParams.appendRow(VariantString(value='*')); // Replace
unnamedParams.appendRow(NIL); // Direction
unnamedParams.appendRow(VariantI2(value=0)); // MatchCase
unnamedParams.appendRow(NIL); // WholeWord
unnamedParams.appendRow(NIL); // PatternMatch
unnamedParams.appendRow(NIL); // SoundsLike
unnamedParams.appendRow(NIL); // FindNext
unnamedParams.appendRow(NIL); // ReplaceOne
unnamedParams.appendRow(VariantBoolean(value=TRUE)); // ReplaceAll
```

In this example, the Microsoft Word for Windows method `EditReplace` has many parameters. If you had specified the parameters using named parameters, then you could have defined and passed in an Array of NamedParameter that contains only 4 NamedParameter objects.

For more information about the `NamedParameter` class and the `Variant` class, see the iPlanet UDS online Help.

Use InvokeMethod or InvokeMethodWithResult to Invoke the OLE Method

To invoke the OLE method, you use a method on your CDispatch object. Each of the following methods invokes an OLE method with the specified name and a list of parameters:

Invoking method	Parameter Type	Return Value
InvokeMethod (methodName=*string*|*TextData*, params=*Array of NamedParameter*)	Named	none
InvokeMethod (methodName=*string*|*TextData*, params=*Array of Variant*)	Positional	none
InvokeMethodWithResult (methodName=*string*|*TextData*, params=*Array of NamedParameter*)	Named	Variant
InvokeMethodWithResult (methodName=*string*|*TextData*, params=*Array of Variant*)	Positional	Variant

The following example shows how you can invoke an OLE method using the `InvokeMethod` method of the `CDispatch` class and a list of named parameters:

```
wordApp.InvokeMethod(methodName='FileOpen', params=namedParams);
```

In this example, the Microsoft Word for Windows method `FileOpen` does not return a value, so we use the `InvokeMethod` of the `CDispatch` class instead of `InvokeMethodWithResult`.

The following example shows how you invoke a method using the `InvokeMethodWithResult` method of the `CDispatch` class and a list of positional parameters:

```
return : Variant = wordApp.InvokeMethodWithResult(
  MethodName='EditReplace',
  Params=unnamedParams);
```

In this example, the Microsoft Word for Windows method `EditReplace` expects a return value, so we use the `InvokeMethodWithResultMethod` of the `CDispatch` class.

NOTE When you import the .pex file, iPlanet UDS removes the quotation marks from the methods. However, when you use a method whose name or whose parameters' names are TOOL reserved words, then you need to specify double quotation marks around the names that are reserved words. To see the list of TOOL reserved words, see the *TOOL Reference Guide*.

For more information about the `InvokeMethod` and `InvokeMethodWithResult` methods, see the iPlanet UDS online Help.

Check the Results of the Method

When you try to retrieve data from some OLE methods, and the data does not exist, the OLE server might return an OLE Variant data type called VT_NULL. In this case, iPlanet UDS sets the IsNull attribute of the Variant object returned or passed back to `TRUE`. You can check the IsNull attribute to see whether the method passed back any data.

Handle Any Exceptions

When you run an application that calls OLE methods, iPlanet UDS can raise the following exceptions:

Exception	Description
OLEInvokeException	Contains information about OLE errors that occur within an invoked OLE method (see the iPlanet UDS online Help). `OLEInvokeException` is a subclass of `OLEException`.
OLEException	Contains information about all OLE errors (see the iPlanet UDS online Help).
UserException	Contains information about errors in the TOOL code (see the iPlanet UDS online Help).

Partition the Client Application

TOOL classes that subclass or instantiate OLE Automation interface classes are restricted classes. These TOOL classes can only run in client partitions running on Windows. Start the Partition Workshop to examine the default configuration for your application, and remove the restricted client partitions from nodes where you do not want these partitions installed.

You can make the distribution just as you would for any regular TOOL project. You can either make the distribution from the Partition Workshop or from within Fscript. For more information about using the Partition Workshop or Fscript, see *A Guide to the iPlanet UDS Workshops* or the *Fscript Reference Guide*.

Install the iPlanet UDS Application

You can install the iPlanet UDS application just like you would any other TOOL client application. For information about installing your iPlanet UDS application, see *iPlanet UDS System Management Guide*.

Chapter 3

Making an iPlanet UDS Service Object an OLE Server

This chapter describes how to make service objects in an iPlanet UDS application available as OLE servers on the Windows 95 and Windows NT platforms.

This chapter also briefly describes how to write an OLE client that accesses an iPlanet UDS service object.

OLE clients can access iPlanet UDS OLE servers that are running on the same machine. If DCOM (Distributed Common Object Model) is available, OLE clients can also access OLE servers that are running on remote machines.

iPlanet UDS service objects that are OLE servers run as local servers, not as in-process servers or in-process handlers. In other words, an iPlanet UDS OLE server starts in its own process space (using the ftexec.exe or the executable for the compiled partition) instead of in the OLE client's process space.

About Making an iPlanet UDS Service Object an OLE Server

iPlanet UDS lets Windows 95 and Windows NT applications use Microsoft's OLE Version 2 (OLE 2) to access data in your iPlanet UDS application.

OLE clients and servers OLE is based on a set of interfaces provided by many Windows applications. A Windows program can interact with the objects associated with another Windows application. These two programs are known, respectively, as the *client* and the *server*. An OLE server is a program that has access to data and that provides functions that might be useful to other programs. An OLE client is a program that obtains this data or interacts with objects associated with the server. A client corresponds to an OLE controller.

This chapter explains how to make an iPlanet UDS service object available to OLE clients as an OLE server.

By making an iPlanet UDS service object available to OLE clients, you can take advantage of the capabilities of the iPlanet UDS runtime system to handle heavy-duty tasks, such as database access, computations, transactions, and so forth, on powerful machines dedicated to these tasks. In fact, these machines can be running non-Windows platforms, such as OpenVMS or UNIX. Meanwhile, you can provide an interface to this application for OLE client applications that are running Windows.

To make a service object available as an OLE server, you can specify that iPlanet UDS generate an ODL (Object Description Language) file and an interface that OLE clients can use to access the service object.

Before you can complete the steps described in this chapter, you need to have the appropriate C++ compiler available on the platforms where you want to compile the shared libraries that make up the OLE automation interface. For information about the C++ compilers supported for each platform, see the *iPlanet UDS System Installation Guide*.

➤ **To make an iPlanet UDS service object available to an OLE client**

1. Define the iPlanet UDS service object that provides functions that you want to make available.
2. Partition the service object to the appropriate nodes.
3. In the Partition Workshop, specify that the service object will be available to an external OLE application.
4. Make the distribution from Fscript or the Partition Workshop to generate the partition startup code, an ODL interface definition file, and a C++ wrapper.
5. Compile and link the C++ wrapper code into a shared library (.DLL) for each platform and compile the ODL file to produce the OLE automation type library. If you can use iPlanet UDS's auto-compile feature, you can perform this step as part of Step 4.
6. Install the shared libraries and type library on the appropriate nodes. You can use the auto-install feature to perform this step as part of Step 4.
7. Start the iPlanet UDS OLE server.

Each step is described in detail starting with "Define a Service Object in an iPlanet UDS Application" on page 69.

Examples

This chapter uses two related examples, called OLEBankEV and OLEBankUV, which are provided in `FORTE_ROOT/install/examples/extsys/ole/server`.

OLEBankEV demonstrates how to define an environment-visible service that acts as an OLE server. OLEBankUV demonstrates how to define a user-visible service object that acts as an OLE server. Both examples show how to write an OLE client application using Microsoft Visual Basic that accesses the iPlanet UDS service object.

For more information about these examples, see "OLEBankEV" on page 350 and "OLEBankUV" on page 351.

Define a Service Object in an iPlanet UDS Application

Defining an iPlanet UDS service object that will be an OLE server is similar to defining any other service object.

There are two limitation to what elements of the service object's class are available to an OLE client.

- An OLE client application can access any methods provided by that service object's class, except for methods that use objects as parameters.
- OLE clients cannot access attributes of an iPlanet UDS service object.

This section describes special issues you need to consider when you define this service object and its class.

Providing an OLE Interface for a Service Object

OLE clients can only pass the following objects and data types as parameters and return values:

TOOL data type	Maps to OLE data type:
boolean	Boolean
CDispatch	IDispatch
CUnknown	IUnknown

TOOL data type	Maps to OLE data type:
double	Double
float	Float
integer	Long
i2	Integer
string	String
ui2	Short
ui4	Long

If an iPlanet UDS service object includes methods that have object parameters or return values other than `CUnknown` and `CDispatch`, these methods are not included as part of the interface to this OLE server.

Providing Methods to Get and Set Attributes

The service object class must provide methods to retrieve or set the values of any attributes in the service object. OLE clients *cannot* assign values to the attributes directly using the "=" operator as in "BankService.MaxClients = 5." Instead, you need to define methods that would set this attribute, as shown:

```
BankService.SetMaxClients(newValue=1);
```

Adding Wrapper Methods to a Service Object

If you are developing an application that you intend to make available as an OLE server, you can define wrapper methods in the service object that accept supported data types as parameters and return values. iPlanet UDS can then export these wrapper methods as part of the OLE interface for this service object.

In the following example, the method invokes a method on the original service object, which returns a BankAccount object. This method then returns a scalar value from the BankAccount object to the OLE client.

```
-- Return the balance of the specified account.
currAcct : BankAccount;
currAcct = BankServer.GetAcctData(acctNumber = Number);
if currAcct.AcctBalance = 0 then
  task.part.logmgr.putline(source =
    'No balance was returned.');
else
  task.part.logmgr.put(source =
    'The balance for account ');
  task.part.logmgr.put(source = Number);
  task.part.logmgr.put(source = ' is ');
  task.part.logmgr.putline(source = currAcct.AcctBalance);
end if;
return currAcct.AcctBalance;
end method;
```

Project: OLEBankUV • **Class:** BankServiceOLEInterface
• **Method:** GetAccountBalance

Defining an OLE Interface in a New Service Object

Depending on the services you want to make available to OLE clients, you might want to define a service object that specifically defines the interface that you want to show to OLE clients. This approach is useful when your application has services running on large, non-Windows machines elsewhere in your environment. You can define methods specifically for this object that call other services in the iPlanet UDS environment.

User-visible service objects If you want to let iPlanet UDS manage communications between your client and server machines, you can define the service object that defines the OLE interface as a user-visible service object that resides in a client partition on the Windows 95 or NT machine.

To define a user-visible service object as an OLE server, create a new project containing a class with a small starting method that accesses the user-visible service object.

This starting method needs to contain an event loop that waits for some indication that the client partition should shut itself down. Without the event loop, the client partition will start and end quickly, before the service object can register itself and before the OLE client application can talk to it. You then need to call this method from your OLE client application to start the OLE server in its iPlanet UDS client partition.

The following example, from the `OLEBankUV.StartupClient.Startup` method, starts the BankServerOLE service object, which is a user-visible service object. This method then goes into an event loop until it receives a Shutdown event.

```
BankServerOLE.Startup();
event loop
  when BankServerOLE.ShutdownEvent do
    exit;
end;
```

Project: OLEBankUV • **Class:** BankServiceOLEInterface • **Method:** Startup

You should also provide a mechanism, such as a `Shutdown` method on the service object, that tells the client partition to shut itself down, as shown in the following example. This method posts an event that is caught by the StartupClient event loop, which causes the task to complete its execution. Because this is a client partition, shutting down the main task also shuts down the user-visible service object.

```
post ShutdownEvent;
```

Project: OLEBankUV • **Class:** BankServiceOLEInterface • **Method:** Shutdown

The OLE client can call this `Shutdown` method to shut down a running iPlanet UDS client, if you do not want to leave the iPlanet UDS client partition running after the OLE client completes running.

NOTE If you define a service object specifically to act as an OLE interface, you need to thoroughly test the methods provided by this service object. It is very difficult to debug problems with your OLE client if this interface is not stable.

Raising Exceptions in the TOOL Code

If you are defining a new service object that interacts with several other iPlanet UDS service objects, you should handle any exceptions that might occur under normal conditions and raise exceptions that contain information that will be useful to the developers of OLE clients. You could, for each exception, define a subclass to `GenericException` that assigns a specific message number to the Message attribute, which the OLE client can use to identify the error condition.

When an iPlanet UDS service object that is being used as an OLE server raises an exception, iPlanet UDS intercepts the exception. iPlanet UDS then defines the values of the ExcepInfo object that is returned to the OLE client. The ExcepInfo object is an OLE structure that contains error information, as described in the following table:

ExcepInfo field	Value
Code	0
DeferredFillIn	NULL
Description	ErrorDesc.Message attribute of raised iPlanet UDS exception
HelpContext	0
HelpFile	‘’
Source	**Forte OLE Automation Service** *export_name* **Method** *method_name*

The Err object in Microsoft Visual Basic corresponds to this ExcepInfo object.

The following example shows how, in the new service object, you can handle an iPlanet UDS exception raised by a server, then raise an exception that is more meaningful to the developer or user of an OLE client. In this example, the `OLEAccountNotFound` class is a subclass of `GenericException` that defines a message number in its Message attribute that the OLE client can check:

```
exception
  when excep : AccountNotFound do
    task.part.logmgr.put(source =
      'Account ');
    task.part.logmgr.put(source = Number);
    task.part.logmgr.put(source = ' was not found.');
    oleExc : OLEAccountNotFound = new();
```

```
    raise oleExc;
```

Project: OLEBankUV or OLEBankEV • **Class:** BankServiceOLEInterface
Method: GetAccountBalance

For information about having an OLE client trap iPlanet UDS exceptions, see "Handling iPlanet UDS Exceptions" on page 101.

Defining the ProgID for the Service Object

As you name the application and service object and set the compatibility level for the application, you should consider that iPlanet UDS automatically generates and registers the progID for the iPlanet UDS service based on the following rules:

distribution_id.*export_name*.**[cl***n***]**

distribution_id is the name of the distribution containing the service object, which is the first 8 characters of the application name.

export_name is the name that OLE clients will use to identify this service object. This name can be set in the Service Object Properties dialog or using the Fscript `SetServiceEOSInfo` command. If no export name is specified, then this name is the name of the project, an underscore (_), and the name of the service object. For more information about the *export_name*, see "Mark a Service Object as an OLE Server" on page 76.

n is the compatibility level for the application. iPlanet UDS automatically generates and registers two ProgID, one with the **cl***n* value and one without, so that users can identify the application without worrying about the release number.

For example, the ProgID for the BankServerOLE user-visible service object in the OLEBankUV project is olebanku.OLEBankUV_BankServerOLE.

Partition the Application Containing the Service Objects

Partitions that contain the service objects that you want to enable as OLE servers must be deployed on nodes that are defined as running either the Windows NT or Windows 95 platform.

Environment-visible service objects When you partition an application containing environment-visible service objects that will be OLE servers, the application is partitioned as usual, so that environment-visible service objects are assigned to server partitions.

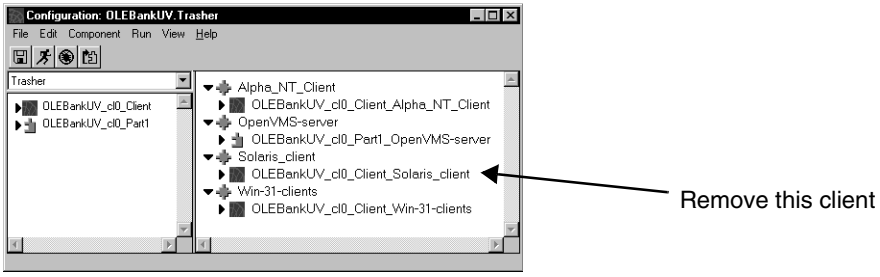
User-visible service objects Partition user-visible service objects in client partitions. Multiple OLE clients can access an OLE server running in a client partition. For more information about designing user-visible service objects as OLE servers, see "Defining an OLE Interface in a New Service Object" on page 71.

NOTE iPlanet UDS assigns partitions to all nodes of the correct type (client or server). You need to remove the assigned partitions containing OLE servers from nodes that do not support OLE; in other words, remove the assigned partitions containing OLE servers from all nodes except those running Windows 95 or Windows NT. If you run a partition containing an OLE server on a node not running Windows 95 or Windows NT, you will get a runtime error.

➤ **To partition your OLE server application**

1. Open the Project Workshop for the main project for your application.
2. In the Project Workshop, choose Run > Partition.

   iPlanet UDS displays the default partitioning for the application in the Partition Workshop.



3. Remove the partition containing the service object that is marked as an OLE server from any nodes that are not running Windows 95 or Windows NT.

You can also use the Fscript command `Partition` to partition the application and the Fscript command `UnassignAppComp` to remove partitions from nodes that are not running Windows 95 or Windows NT.

Server application If the service object that is marked as an OLE server is environment-visible, you can configure your application as a server application in the Project Workshop using the File > Configure as Server command.

For information about partitioning applications, see *A Guide to the iPlanet UDS Workshops* or the *Fscript Reference Guide*.

Mark a Service Object as an OLE Server

To mark a service object as an OLE server, you must set the external type for the service object as OLE. You can optionally set the name that OLE clients will use to identify this service object. You can mark the service object using either the Partition Workshop or the Fscript command `SetServiceEOSInfo`.

➤ **To mark a service object in the Partition Workshop**

1. Open the Service Object Properties dialog for the service object you want to make available to OLE clients.
2. Click the Export tab to go the Export page.
3. Set the value of the External Type field as `OLE`.
4. Set the Export Name field to the name that OLE clients will use to identify this service object. This value is a text string that has a letter as its first character.

   If no export name is specified, then the OLE server name is the project name for the service object, an underscore character (_), and the service object name. For the exact syntax of the server entry in the registry, see "Defining the ProgID for the Service Object" on page 74.
5. Click the OK button.

For more information about specifying service object properties, see *A Guide to the iPlanet UDS Workshops*.

➤ **To mark a service object using the Fscript command SetServiceEOSInfo**

1. Start Fscript.
2. Open the repository, make the deployment environment the current environment, and make the main project for the application containing this service object the current plan, using a series of Fscript commands like the following:

```
fscript> FindEnv MyEnvironment
fscript> FindPlan MainProject
```

3. Partition this application using the `Partition` command.
4. Enter the `SetServiceEOSInfo` command using the following syntax:

   SetServiceEOSInfo *service_object_name* **OLE** [*export_name*]

   service_object_name is the name of the service object that you want to make available to the external object service. If the current project contains the service object, you can specify just the name of the service object; otherwise, *service_object_name* should specify the project name and the service object name, like BankServices.BankServer.

   export_name is the name that OLE clients will use to identify this service object. This value is a text string that has a letter as its first character. The length of the export name depends on your particular implementations of OLE Automation. If no export name is specified, then the OLE server name is the project name for the service object, an underscore character (_), and the service object name. For the exact syntax of the server entry in the registry, see "Defining the ProgID for the Service Object" on page 74.

   The following example shows how you could mark a service object as an OLE server:

```
fscript> SetServiceEOSInfo OLEBankUV.BankServerOLE OLE
BankServer
```

   In this example, OLEBankUV is the name of the project, and BankServerOLE is the name of the service object. BankServer is the name that will be registered for this OLE server.

   For more information about Fscript, see *Fscript Reference Guide*.

5. Use the `Partition` command again to repartition the application based on the new settings on the service object.
6. Use the `UnassignAppComp` command to remove the partition containing the service object that is an OLE server from nodes that are not running Windows.

Make the Distribution

Make a distribution using the Partition Workshop or the Fscript `MakeAppDistrib` command.

If your environment is set up for auto-compiling, you can compile, link, and install the shared libraries and type libraries on the appropriate nodes when you make the distribution.

If you choose not to use the auto-compile and auto-install features, you can find the steps for compiling, linking, and installing the shared libraries and type libraries without using the automated features, in the following sections:

- "Compile and Link to Produce a Shared Library and Type Libraries" on page 81
- "Install the Executable" on page 84

Making the Distribution with Auto-Compile and Auto-Install

Your iPlanet UDS system manager can set up your system so that you can automatically compile and link the code that was generated into shared libraries.

If you use the auto-compile and auto-install features to compile, link, and install the shared libraries and type libraries, skip to "Start the iPlanet UDS Partition" on page 85.

The steps for setting up the system to enable auto-compile and auto-install are explained in *iPlanet UDS System Management Guide*. In general, your system manager must set up the following components on your system:

- one or more code generation servers to generate the code for the distribution
- a server that manages how and where shared libraries are compiled and linked

- one auto-compilation server for each platform where the shared libraries and type libraries will be installed. Each of these servers must have access to the C++ compiler for that platform.

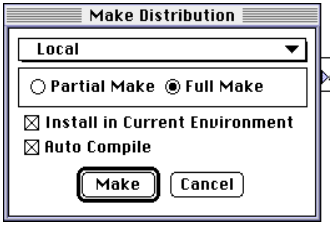
If your system manager has set up these components, then you can make the distribution with the auto-compile feature. This feature performs the following steps automatically, in addition to the steps that are performed for the TOOL application that contains the service object:

- generate an ODL file
- compile the ODL file into a type library
- generate C++ wrapper code
- compile and link the C++ wrapper code into the shared library required for each platform
- place the shared libraries, the ODL file, and the type libraries into the appropriate distribution directories

If you also selected auto-install, making the distribution also installs the shared libraries on the appropriate nodes in the development environment, according to the configuration you specified when you partitioned your TOOL application.

➤ **To make a distribution with auto-compile and auto-install**

1. After you have partitioned your iPlanet UDS application, choose the File > Make Distribution command.
2. In the Make Distribution dialog, select Partial Make (to update a distribution) or Full Make (to create a new distribution), then select the toggles for Install In Current Environment and Auto Compile.



3. Select the Make button.

   This step generates the needed files, compiles the files, then copies them to the appropriate FORTE_ROOT\userapp directories on the nodes where they are assigned. The files that are most important for the OLE server are `ole_#.dll`, `so#.odl` and `so#.tlb`, where # is an arbitrary number generated by iPlanet UDS. The `.tlb` file is the type library for the OLE server.

You can also use the `MakeAppDistrib` command in Fscript to make a distribution with auto-compile and auto-install, as shown:

```
fscript> MakeAppDistrib 1 "" 1 1
```

For more information about making a distribution, see *A Guide to the iPlanet UDS Workshops*. For information about the Fscript `MakeAppDistrib` command, see the *Fscript Reference Guide*.

At this point, skip to "Start the iPlanet UDS Partition" on page 85.

Making the Distribution without Auto-Compiling

If you are making the distribution without using the auto-compile feature, then this step generates code for the current configuration of the TOOL application and the ODL file for each service object being defined as an OLE server. You need to compile the ODL file to produce the type library for the OLE server, then compile and link the C++ wrapper code to produce the shared library. Then, you need to place the shared library in the appropriate distribution directories to enable the iPlanet UDS system manager to automatically install the shared library.

Making a distribution produces the following items in the distribution directory in addition to the usual files for the partition:

.bom file Describes the files to be compiled for the OLE server.

C++ module Contains the generated C++ code that makes the iPlanet UDS service object available to an OLE client. Registers the iPlanet UDS services provided by the service object in the Windows registry and the COM library.

.odl file Describes a marked service object that is available to OLE clients. iPlanet UDS exports only the methods in the service objects that do not contain unsupported types as parameters or return values. An ODL file is generated for each marked service object. This ODL file is compiled into a type library when you use the **fcompile** utility.

For example, the files generated for the OLEBankUV application could be `ole_0.bom`, `s01.cc`, and `so1.odl`.

After making the distribution, iPlanet UDS puts these files in the distribution directory:

FORTE_ROOT/appdist/*environment_id*/*distribution_id*/cl#/codegen/partition_id

Directory Name	Description
environment_id	First 8 characters of the environment name where you want your application to be installed.
distribution_id	First 8 characters derived from the name of the main TOOL project for the application containing this service object.
cl#	Compatibility level for this project, as specified for the main TOOL project for the application containing this service object.
partition_id	The first 6 characters of the application name plus the partition number.

If any of the partitions in your application are compiled partitions, this directory also contains the .pgf files for these partitions.

For example, if you make a distribution on Windows NT, the files for the OLEBankUV example would be in the `FORTE_ROOT\appdist\centrale\olebanku\cl0\codegen\oleban0 directory`.

Compile and Link to Produce a Shared Library and Type Libraries

This section describes the steps you need to perform if you are not using iPlanet UDS's auto-compile feature, described in "Making the Distribution with Auto-Compile and Auto-Install" on page 78.

fcompile command

iPlanet UDS provides an `fcompile` command that lets you generate a shared library for the wrapper code and ODL files on a given platform. This utility compiles the ODL files to produce the OLE automation type libraries. The `fcompile` command then compiles and links the C++ module into a shared library.

If any partitions for your application are compiled partitions, the `fcompile` command, by default, also generates code and compiles and links the compiled partitions from their .pgf files in the **`codegen/`***partition_id* directory.

The syntax of the `fcompile` command when you compile parts of an application that include a service object to be made available as an OLE server is:

Portable syntax (all platforms)

fcompile [**-c** *component_generation_file*] [**-d** *target_directory*]
[**-cflags** *compiler_flags*] [**-lflags** *linking_flags*]
[**-fm =** *memory_flags*] [**-fl =** *logger_flags*] [**-cleanup**]

The following table describes the command line flags for the `fcompile` command:

Flag	Description
-c *component_generation_file*	Specifies the file that iPlanet UDS compiles. This value includes the path where the file resides if the file is not in the current directory. By default, iPlanet UDS compiles all files in the current directory.
-d *target_directory*	Specifies where the compiled directories will be placed. By default, `fcompile` compiles files in the current directory, and places the compiled files in the current directory. `target_directory` is a directory specification in local syntax. If the **-c** flag is also specified, the **-d** flag specifies only where the compiled component files will be placed. Otherwise, the directory specified by the **-d** flag specifies both the directory containing the files to be compiled and the directory where the compiled files will be placed.
-cflags *compiler_flags*	Specifies any C++ compiler options.
-lflags *linking_flags*	Specifies any linking flags.
-fm *memory_flags*	Specifies the space to use for the memory manager. See *A Guide to the iPlanet UDS Workshops* for information.
-fl *logger_flags*	Specifies the logger flags to use for the command. See *A Guide to the iPlanet UDS Workshops* for information.
-cleanup	Deletes all the files except for the newly compiled shared libraries.

Steps for Compiling and Linking

➤ **To compile C++ wrapper code and ODL files**

1. Copy all the files that you generated by making the distribution to a node that has the platform on which the partition containing the service object will be installed. This node must have the required C++ compiler and ODL files so that the code can be compiled and linked.

   The files you need to copy are in the directory path described under "Make the Distribution" on page 78.

2. Use `fcompile` to generate, compile, and link code into shared libraries.

   For example, if you have copied all the files in the `codegen` directory to the node to be compiled, then you could just enter the `fcompile` command with no parameters.

   If you only want to compile the partition or just the wrapper code and OLE files, use the `fcompile` command with its **-c** flag. To compile just the partition code, specify the .pgf file with the **-c** flag. To compile the OLE file, specify the .bom file with the **-c** flag.

3. Create the following distribution directories and copy the shared library (.dll) and the type libraries (.tlb) to this distribution directory:

   `FORTE_ROOT/appdist/`*environment_id/distribution_id/cl#/platform/partition_id*

Directory Name	Description
environment_id	First 8 characters of the environment name where you want your application to be installed.
distribution_id	First 8 characters derived from the name of the main TOOL project for the application containing this service object.
cl#	Compatibility level for this project, as specified for the main TOOL project for the application containing this service object.
platform	Architecture name for the platform where this shared library will be installed, for example, PC_WIN.
partition_id	The first 6 characters of the application name plus the partition number.

For example, assume that the distribution is on Windows NT and you have compiled files for the OLEBankUV application for Windows NT. You would create a `pc_nt\oleban0\` directory under the `cl0` subdirectory shown in "Making the Distribution without Auto-Compiling" on page 80 and copy all the files there. At the end of this step, the `ole_1.dll`, `ole_1.lib`, `ole_1.exp`, and `so1.tlb` should be in the `FORTE_ROOT\appdist\centrale\olebanku\cl0\pc_nt\oleban0\`.

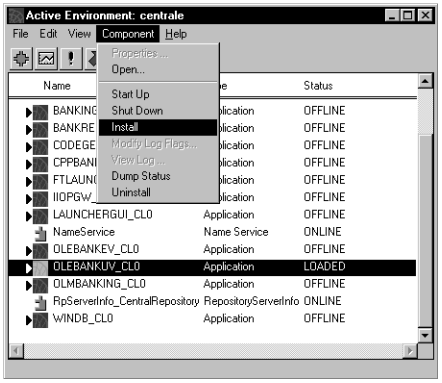
Install the Executable

This section describes the steps you need to perform if you are not using iPlanet UDS's auto-install feature, described in "Making the Distribution with Auto-Compile and Auto-Install" on page 78.

Using the Environment Console or Escript, install the application containing the iPlanet UDS service object using iPlanet UDS installation procedures.

➤ **To install the iPlanet UDS application**

1. In the Environment Console, choose File > Load Distribution.
2. In the Load Distribution dialog, select the node on which you made the distribution for this library, then select the application distribution.
3. Choose View > Application Outline.
4. Select the application distribution that you just loaded, then choose Component > Install.



You can also use Escript commands to install an iPlanet UDS application, as shown:

```
escript> LoadDistrib OLEBankUV cl0
escript> Install
```

For more information about installing applications in iPlanet UDS, see *iPlanet UDS System Management Guide*.

Start the iPlanet UDS Partition

You must start the iPlanet UDS partition that contains service objects that are OLE servers at least once so that the partition can register its service objects in the Windows registry. If you have the registry open when you start this partition, you need to refresh the Registry Editor window to see the iPlanet UDS service object.

An iPlanet UDS partition performing as an OLE server behaves like other iPlanet UDS partitions. If it is a server partition, you can start it using the Environment Console or Escript. You can also autostart iPlanet UDS server partitions using iPlanet UDS calls. If this partition is a client partition, it also behaves like other iPlanet UDS client partitions, except that they can be autostarted by an OLE client.

An OLE client can autostart the iPlanet UDS server partition if the partition has already been registered as an OLE server in the Windows registry. By default, the interpreted service partition is autostarted. If you want to have the compiled service partition autostarted, see "Modifying How a Partition Is Autostarted" on page 89.

If an OLE client tries to access the OLE server before the iPlanet UDS server partition has registered the service object in the Windows registry, you will get an error.

NOTE When an OLE client invokes a request on an iPlanet UDS OLE server, the iPlanet UDS service object waits for the OLE request to complete processing before it processes any other requests from an OLE client. However, the iPlanet UDS service object can continue to process requests from iPlanet UDS clients.

Registering the Partition

To have OLE Automation recognize a started partition as an OLE server, start the partition in one of the following ways:

- Start the partition using the `ftexec` command or its compiled executable at a command prompt, as shown in the following example for a server partition:

  ```
  ftexec -ftsvr 0 -fi bt:c:\forte\userapp\olebanku\cl0\oleban0
  ```

- If the partition is a client partition, start it using the generated icon, which starts the client partition through the Launch Server using the `ftcmd run` command
- Have the OLE client call the OLE server, which makes OLE Automation auto-start the OLE server.

  What actually happens in this case is that OLE Automation invokes an `ftexec` command to start the partition as interpreted, or runs the compiled executable, depending on how you set up the registry. For more information about locating and changing the command used to auto-start the partition, see "Modifying How a Partition Is Autostarted" on page 89.

Troubleshooting the OLE Server

This section describes some basic troubleshooting techniques that you can use when an OLE server does not work properly.

If the OLE server does not register or advertise itself when you start it up, you should check the log file or the trace window for the partition to make sure that the partition has registered and advertised itself successfully. The log file should contain messages like the following if the partition has successfully advertised itself as an OLE server:

```
OLEBanking_BankServerOLE: Registering service ...
OLEBanking_BankServerOLE: Application ID : %{FORTE_ROOT}/userapp/oletest/cl0


OLEBanking_BankServerOLE:   Partition ID     : ole_1
OLEBanking_BankServerOLE: Interface Name : OLEBanking_BankServerOLE

OLE LSTN: Forte partition OLE enabled ...
Successfully completed OLE advertisement for OLEBanking_BankServerOLE
```

If you instead find errors about the .dll not being found, make sure that you have correctly partitioned and compiled the `ole_#.dll` file, as described in "Make the Distribution" on page 78 and "Compile and Link to Produce a Shared Library and Type Libraries" on page 81.

NOTE If you run a partition containing an OLE server on a node not running Windows 95 or Windows NT, you will get a runtime error.

Customizing Registry Entries for an iPlanet UDS OLE Server

This section describes how to modify Windows registry entries to:

- delete obsolete entries from the Windows registry
- specify how a partition is autostarted

Deleting Obsolete Entries from the Windows Registry

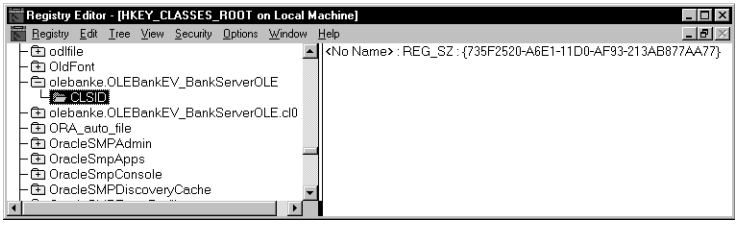
When you uninstall a partition or an application that contains a partition that is an OLE server, you need to remove the entry for this OLE server from the Windows Registry. Otherwise, other applications that rely on the information in this registry might mistakenly assume that this OLE server is still available.

➤ **To remove obsolete entries from the Window Registry**

1. In the HKEY_CLASSES_ROOT section of the registry, find the ProgID entries for the OLE server. For example, the ProgIDs for a service object could be olebanke.OLEBanking_BankServerOLE and olebanke.OLEBanking_BankServerOLE.cl0. iPlanet UDS always registers the service object with names that do and do not include the compatibility level, so that a client application can choose a particular release of a product, but does not need to.

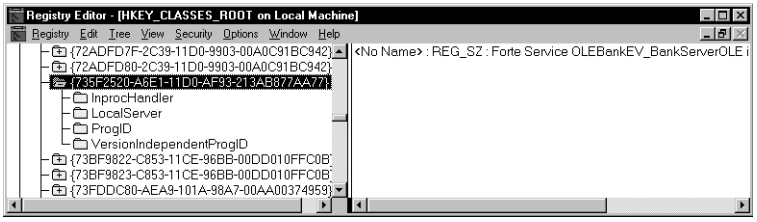
   For information about determining the ProgID for an OLE server, see "Defining the ProgID for the Service Object" on page 74.

2. Double-click on the ProgID entry and the CLSID entry to determine the CLSID for the OLE server.



Make a note of this CLSID, which is a unique identifier string containing 32 hex digits enclosed in braces, such as {735F2520-A6E1-11D0-AF93-213AB877AA77}.

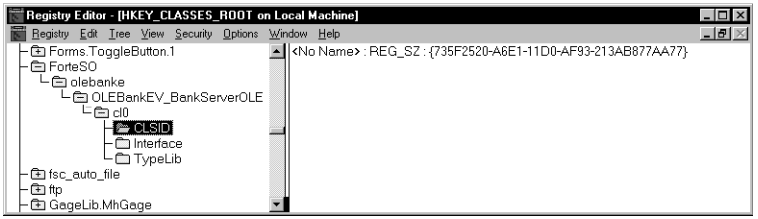
3. In the HKEY_CLASSES_ROOT section of the registry, double-click on the CLSID folder, and locate the folder labeled with the CLSID for the OLE server, for example, {735F2520-A6E1-11D0-AF93-213AB877AA77}.



4. Delete the CLSID entry.

5. Locate the ProgID entries for the iPlanet UDS service object again and delete them.

6. Find the file labelled ForteSO. The files in this folder are arranged hierarchically so that the file names, separated by dots, actually represent a hierarchy of folders containing other folders. For example, if the ProgID for a service object is olebanke.OLEBanking_BankServer.cl0, ForteSO contains a folder named olebanke, which contains a folder called OLEBanking_BankServer, which in turn contains a folder called cl0.



7. Delete the folders corresponding to the application or partition that contains the OLE server. For example, if you uninstall the OLEBankEV application, you should delete the folder olebanke, which also deletes all the folders it contains.

Modifying How a Partition Is Autostarted

An OLE client can autostart the iPlanet UDS server partition if the partition has already been registered as an OLE server in the Windows registry. By default, the interpreted service partition is autostarted with the system defaults. If you want the OLE client to auto-start the compiled server partition instead or set flags on the `ftexec` or executable command, you need to change the entry in the registry, as described in this section.

➤ **To change how a partition is auto-started**

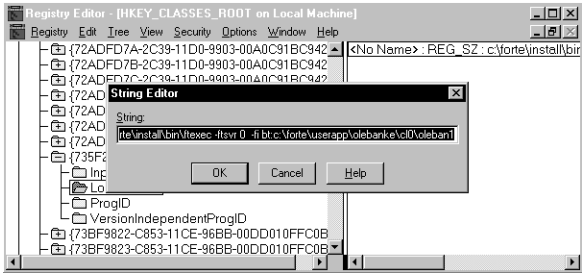
1. In the HKEY_CLASSES_ROOT section of the registry, find the ProgID entries for the OLE server. For example, the ProgID for a service object could be olebanke.OLEBanking_BankServerOLE.

   For information about determining the ProgIDs for an OLE server, see "Defining the ProgID for the Service Object" on page 74.

2. Double-click on the ProgID entry and the CLSID entry to determine the CLSID for the OLE server.

   Make a note of this CLSID, which is a unique identifier string containing 32 hex digits enclosed in braces, such as {735F2520-A6E1-11D0-AF93-213AB877AA77}.

3. In the HKEY_CLASSES_ROOT section of the registry, double-click on the CLSID folder, and locate the folder labeled with the CLSID for the OLE server, for example, {735F2520-A6E1-11D0-AF93-213AB877AA77}.
4. Open the Local Server folder for the CLSID entry.
5. Double-click on the key value in the right half of the editor window.



6. Edit the command to contain the command and flags that you want the partition to use when it auto-starts.

   For example, suppose the original key value, which specifies that OLE Automation should auto-start an interpreted partition looks like the following (all in one line):

```
c:\forte\install\bin\ftexec -ftsvr 0 -fi
bt:c:\forte\userapp\olebanke\cl0\oleban0
```

   To have OLE Automation start a compiled partition with a logger flag, specify a command like the following:

```
c:\forte\userapp\olebanke\cl0\oleban0 -fl 'err.log(err:user)'
```

Using DCOM with iPlanet UDS OLE Servers

This section describes how to use the (Distributed Component Object Model) DCOM feature of Microsoft OLE with your iPlanet UDS OLE servers. You can use DCOM on Windows NT and on Windows 95 with DCOM 95.

When you make an iPlanet UDS service object available as an OLE automation server, you can also set up the service object to be accessed by remote clients. You need to set up the clients and the server to enable the clients to access the iPlanet UDS OLE automation server, as described in this section.

If you can access your iPlanet UDS OLE server using COM on a single machine, you should be able to use DCOM to access your iPlanet UDS OLE server by following the steps in this section. You should ensure that you can run your OLE client with the iPlanet UDS OLE server on the same machine before you try to run any remote clients with the iPlanet UDS OLE server.

Windows security and DCOM configurations If your server machine permissions, user profiles, and DCOM configurations are not set up properly, your client applications will not be able to access the iPlanet UDS OLE server. You should refer to the Windows NT, Windows 95, and DCOM documentation provided by Microsoft to ensure that you have set up your user profiles, file sharing, and DCOM configuration correctly for your server machine and your iPlanet UDS OLE server.

This section will use the OLEBankUV example to illustrate the various steps.

➤ **To use DCOM with your iPlanet UDS OLE servers**

1. Deploy your iPlanet UDS application as an iPlanet UDS OLE server.

   See "About Making an iPlanet UDS Service Object an OLE Server" on page 67.

2. Start the iPlanet UDS partition.

   Start the iPlanet UDS partition that contains service objects that are OLE servers at least once so the partition can register its service objects in the Windows registry.

   At this point, iPlanet UDS also generates a .reg file that contains information needed to register the identity and location of your iPlanet UDS OLE server on the clients that will use this OLE server. See "Customizing Registry Entries for an iPlanet UDS OLE Server" on page 87.

   This file has the same name as the service object, and is in the same `FORTE_ROOT\userapp` subdirectory as the partition containing the service object.

3. Change the server security settings.

   On the machine running the iPlanet UDS OLE server, you need to change the security settings to permit remote clients to start or access the iPlanet UDS OLE server. This step is described in "Changing Security Settings."

4. Register the iPlanet UDS OLE server on client machines.

   You need to register the iPlanet UDS OLE server on the client machine to enable the client machines to locate and access the OLE server. You need to perform this step for each client machine that will access the iPlanet UDS OLE server. This step is described in "Registering the iPlanet UDS OLE Server on Client Machines" on page 99.

Changing Security Settings

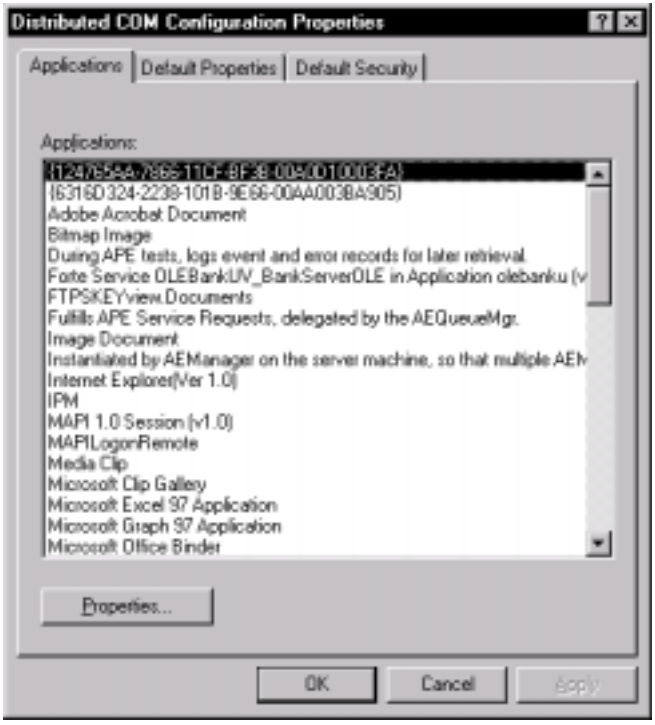
The Microsoft Distributed COM Configuration Properties utility lets you change the security settings for all OLE servers on your machine, or for specific OLE servers.

CAUTION The security settings shown in the section are used to explain what settings you need to be aware of, and to demonstrate settings that will work with our example. You need to set your security settings to follow your own security policies.

➤ **To start the Distributed COM Configurations Properties utility**

1. Locate and start the dcomcnfg.exe utility on the machine running the iPlanet UDS OLE server.

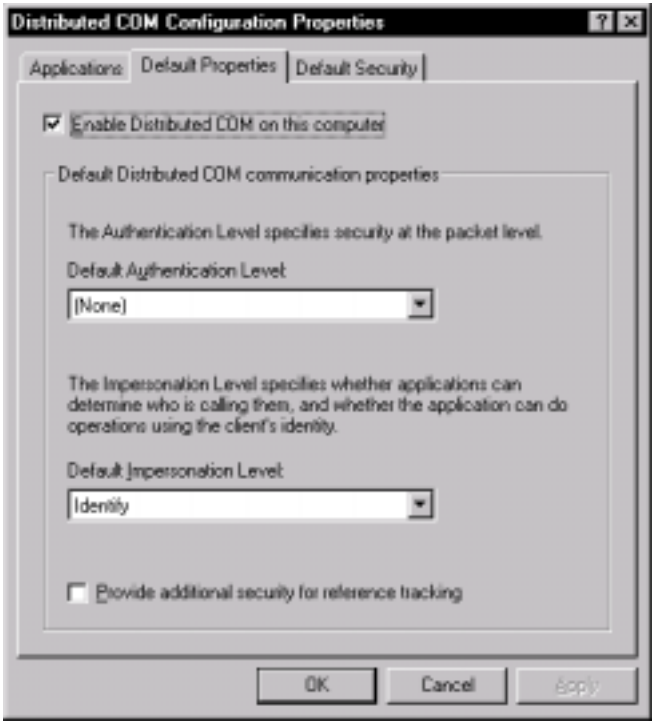
   The Distributed COM Configuration Properties dialog opens.



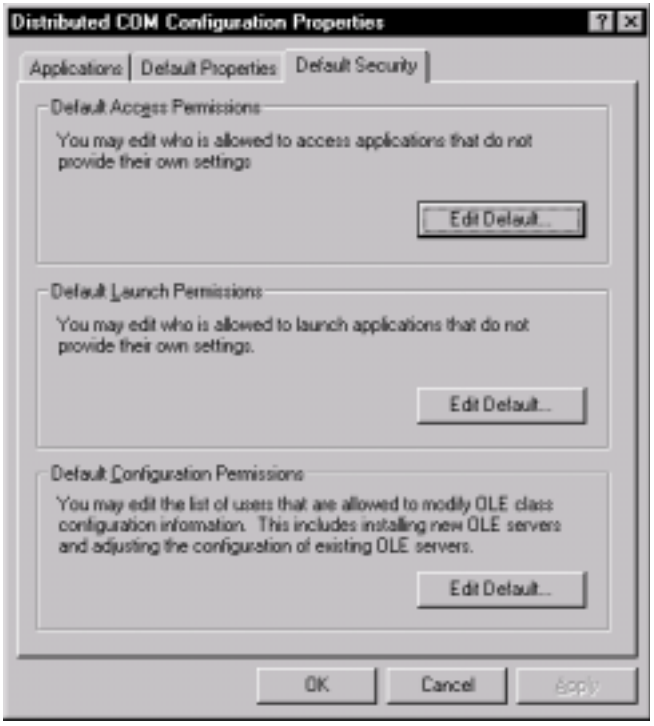
➤ **To change the security settings for all OLE servers on your machine**

1. Start the Distributed COM Configurations Properties utility, as explained in Step 1 on page 93.
2. Choose the Default Properties tab page.
3. Change the Default Authentication Level.

   For example, you might want to set this field to (None), if you do not want any security-checking to occur on communications between applications.

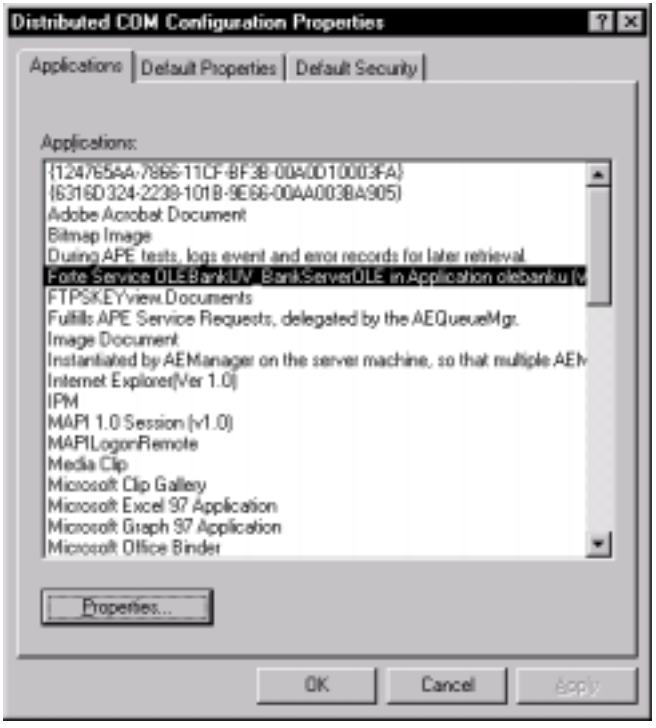


You can also choose the Default Security tab page and change the Default Launch Permissions.

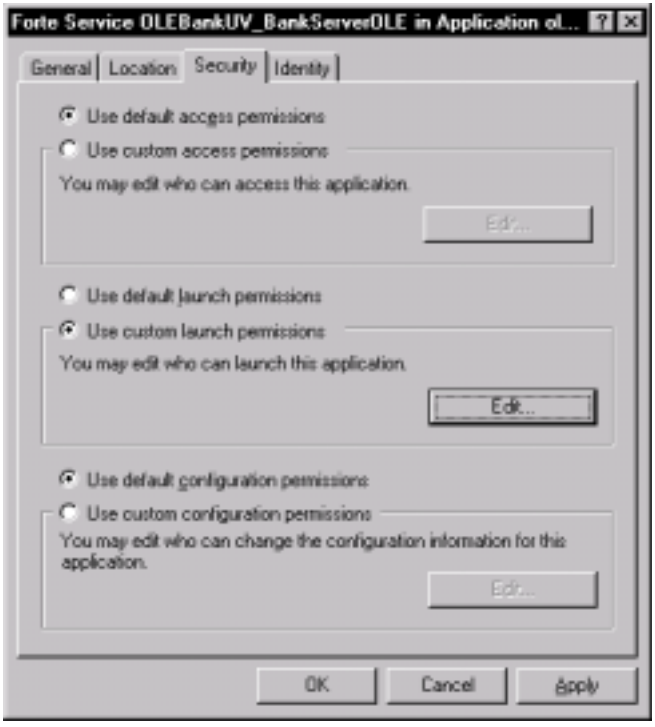


➤ **To change the security settings for a specific OLE server**

1. Start the Distributed COM Configurations Properties utility, as explained in Step 1 on page 93.
2. Choose the Applications tab page.
3. Choose the Properties button.

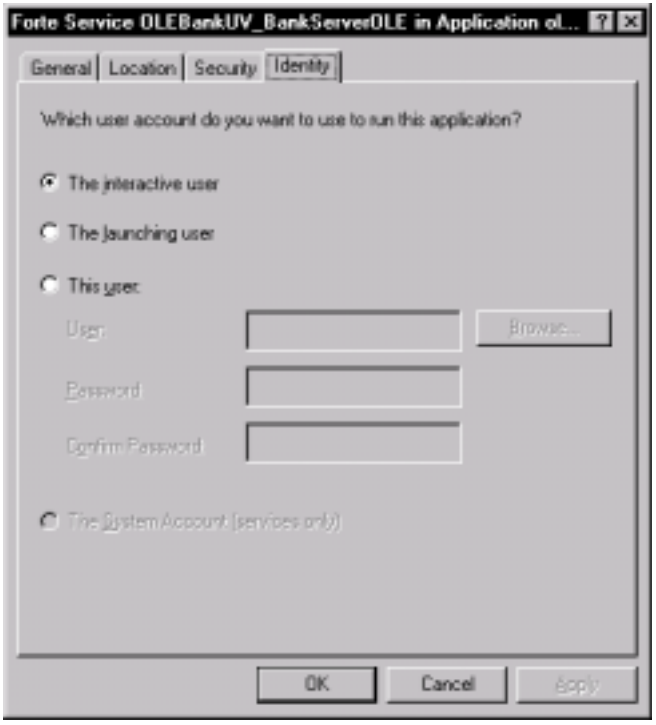


4. In the application properties dialog, choose the Security tab page to customize the access permissions, custom launch permissions, or configuration permissions.



5. You can also choose the Identity tab page to choose the user account that will be associated with this application.

   For example, we chose the interactive user.



Registering the iPlanet UDS OLE Server on Client Machines

You need to register the iPlanet UDS OLE server on the client machine to enable the client machines to locate and access the OLE server. You need to perform this step for each client machine that will access the iPlanet UDS OLE server.

➤ **To register the iPlanet UDS OLE server on a client machine**

1. Copy the .reg file generated for the iPlanet UDS OLE server onto the client machine, or remotely mount the drive containing the .reg file and locate it from the client machine.
2. Use the .reg file to register the iPlanet UDS OLE server in the client machine's Windows registry. To perform the registration, you can do one of the following:

   Double-click the icon for the .reg file. The information in the file is automatically added to the Windows registry.

   - Start the regedit.exe utility, then import the .reg file using the Registry > Import Registry File command.

Writing OLE Clients That Access an iPlanet UDS Service Object

An OLE client that accesses an iPlanet UDS OLE server can be written using the same approach that you would use for any other OLE client.

OLE clients can access iPlanet UDS OLE servers that are running on the same machine. If DCOM (Distributed Common Object Model) is available, OLE clients can also access OLE servers that are running on remote machines.

Before you can write an OLE client that accesses an iPlanet UDS service object, you need the type library file (.tlb). This file defines what methods on the iPlanet UDS service object are accessible to an OLE client through OLE automation. This type library file is the result of compiling the ODL file that was generated when you made a distribution for the application containing the service object.

In a product like Microsoft Visual Basic, you can reference the .tlb file to make your OLE server known to your Visual Basic OLE client.

The OLE client needs to know the server name of the iPlanet UDS OLE server. The iPlanet UDS OLE server advertises itself using the Windows registry. "Determining the ProgID for the Service Object" describes how you can find the ProgID.

If no export name was specified for the service object, then the iPlanet UDS OLE server advertises itself using the project name for the service object, an underscore character (_), and the service object name. For example, if the project name is Taxes, and the service object is named Calculations, then the *server_name* value is Taxes_Calculations.

iPlanet UDS OLE servers return ExcepInfo objects to an OLE client. This ExcepInfo object contains the information described in "Raising Exceptions in the TOOL Code" on page 73.

Determining the ProgID for the Service Object

The progID for the iPlanet UDS service that you are accessing from an OLE client is:

distribution_id.*export_name*.[**cl***n*]

distribution_id is the name of the distribution containing the service object, which is the first 8 characters of the application name.

export_name is the name that OLE clients will use to identify this service object. This name can be set in the Service Object Properties dialog or using the Fscript `SetServiceEOSInfo` command. By default, this name is the name of the project, an underscore (_), and the name of the service object. For more information about the *export_name*, see "Mark a Service Object as an OLE Server" on page 76.

n is the compatibility level for the application. For a release independent ProgID, do not specify the **cl***n* extension.

For example, the ProgID for the BankServerOLE user-visible service object in the OLEBankUV project is olebanku.OLEBankUV_BankServerOLE.

Handling iPlanet UDS Exceptions

When an iPlanet UDS service object that is being used as an OLE server raises an exception, iPlanet UDS intercepts the exception. iPlanet UDS then sets the values of the ExcepInfo object that is returned to the OLE client. The following table shows how the ExcepInfo object values represent a raised iPlanet UDS exception:

ExcepInfo field	Value
Code	0
DeferredFillIn	NULL
Description	ErrorDesc.Message attribute of raised iPlanet UDS exception
HelpContext	0
HelpFile	‘’
Source	**Forte OLE Automation Service** *export_name* **Method** *method_name*

The Err object in Microsoft Visual Basic corresponds to this ExcepInfo object.

The following example shows how you could trap an iPlanet UDS exception in Visual Basic:

```
On Error GoTo CheckError
  acctNumber = txtAcctNumber.Text
    If boolStarted = False Then
      Set BankServiceObject =
        CreateObject("OLEBanke.OLEBankEV_BankServerOLE")
  boolStarted = True
    End If
. . .
CheckError:
  If Err.Number <> 0 Then
    If Err.Description = conAccountNotFound Then
      MsgBox "There is no account with account number " _
        & acctNumber & ". Valid account numbers are _
        1000, 2000, and 3000."
        , vbExclamation
      ClearFields
    ElseIf Err.Description = conGenericException Then
      MsgBox "An unexpected Forte Exception occurred: Error " _
        & Err.Number & ": " & Err.Description _
        & " - " & Err.Source, vbExclamation
    Else
      MsgBox "Error " & Err.Number & ": " & Err.Description _
        & " - " & Err.Source, vbExclamation
    End If
  End If
```

Chapter 4

Using ActiveX Controls in TOOL Applications

Microsoft defines a specification for ActiveX controls that lets programmers design controls that can be used in a Windows graphical user interface. These controls are also called OCX controls or OLE custom controls.

This chapter explains how you can use ActiveX in the graphical user interfaces of your iPlanet UDS clients that are running in a Windows NT or Windows 95 environment.

These ActiveX controls are specialized kinds of OLE servers, so many aspects of using ActiveX controls are similar to the steps for using external OLE applications.

About Using ActiveX Controls in TOOL Applications

This chapter explains how you can use ActiveX controls in the graphical user interfaces of your iPlanet UDS clients that are running in a Windows NT or Windows 95 environment.

This chapter assumes that you want to use the default dispatch interface and events, and that a control defines only one dispatch interface and one event set for the control. If you do not know whether you are using the default dispatch interface and events or not, you probably are.

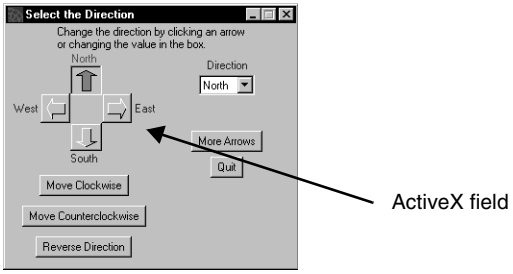
This chapter also assumes that you have documentation available for the control you are using so that you can determine how to use the control's methods, properties, and events as intended.

If you need to use dispatch interfaces that are not the default dispatch interface or non-default event sets, see iPlanet UDS Technote 10825.

Overview

You can use ActiveX controls in the windows of your iPlanet UDS client application. Using predefined controls can save you the work of developing complex controls yourself.

When you use an ActiveX control in your iPlanet UDS window, you need to use an ActiveXField widget to contain the ActiveX control. The ActiveX control is actually a mapped type of the ActiveXField widget.



Your iPlanet UDS application is the container for the control. Your application can access the methods and properties of the control, and the control can send events, using the OLE event mechanism, to your application. This event mechanism automatically calls methods defined in the ActiveX interface class that have the same names as the control's events.

Support for ActiveX Controls

iPlanet UDS does not support ActiveX controls that can contain other unrelated ActiveX controls, and therefore have "nested" interfaces. For example, iPlanet UDS does not support ActiveX controls that are grid fields or panels which can contain buttons, graphics, and documents, each with their own interfaces. iPlanet UDS cannot interact with ActiveX controls within other ActiveX controls.

This limitation does not affect the complexity of the controls that you can use, as long as the ActiveX control provides one interface for all its components. For example, you can interact with calendar or spreadsheet ActiveX controls.

Including ActiveX Controls in TOOL Applications

Your TOOL application can use an ActiveX control in two ways:

- as though it were another iPlanet UDS widget, with complete support for methods and events
- primarily as a display of information, with limited support for methods and no ability to catch events

Using ActiveX Controls as Widgets

The steps for using ActiveX controls as fully-functional widgets are complex; however, these steps provide complete support for the functions and events available for the ActiveX controls.

To most effectively use an ActiveX control in your iPlanet UDS application, you need to install the ActiveX control on your system, then use the `Olegen` utility to generate a .pex file that defines a TOOL interface to the ActiveX control. You then need to import the generated .pex file into your workspace. These steps are explained thoroughly starting with "Producing TOOL Classes For an ActiveX Control" on page 106.

When you use an ActiveX control as a widget, you define an ActiveXField widget and specify an ActiveX interface class for an ActiveX control as the mapped type for the ActiveXField widget. You can then invoke methods and access properties of the control. Your TOOL code can also handle events sent by the ActiveX control. These steps are described in "Developing an iPlanet UDS Application that Uses ActiveX Controls" on page 109.

Using ActiveX Controls to Display Information

If you want to include an ActiveX control in an iPlanet UDS window to display information but not interact with it, you can simply install the ActiveX control, then insert the ActiveX control in an ActiveX field in your application. In this case, however, you can only interact with the control using its CDispatch interface, and you cannot handle events sent by the control.

ActiveX controls are a type of OLE automation server. If you want to use the functions for the ActiveX control, you need to interact directly with the control's CDispatch interface using the OLE library's `CDispatch` class. This is the same approach that you can use to interact with OLE server applications. For information, see "Invoking Methods on OLE Interfaces Using CDispatch" on page 60.

Examples

The examples used in this chapter are based on the ActiveXDemo sample application provided in the `FORTE_ROOT\install\examples\ole\server` directory. This example uses a simple ActiveX control provided by iPlanet UDS called FourDir.

For more information about the example, see "ActiveXDemo" on page 337.

Producing TOOL Classes For an ActiveX Control

This section describes how to use the `Olegen` utility to generate TOOL classes that let you interact with ActiveX controls.

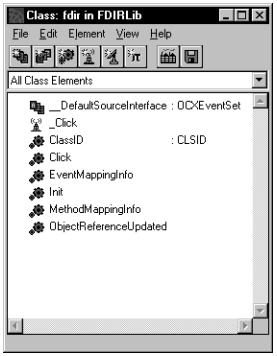
To interact with an ActiveX control from your TOOL code, you need to have TOOL classes that correspond to that control in your development repository.

`Olegen` generates, at a minimum, the following types of classes for each ActiveX control:

default dispatch class A subclass of the `CDispatch` class of the OLE library. This class provides the methods and attributes for the ActiveX control.

ActiveX interface class A subclass of the default dispatch interface class for the control. The ActiveX interface class inherits the attributes and methods of the default dispatch interface. In addition, this class defines the events that can be sent by the control and the methods that handle these events. To see all the methods, attributes, and events available in the ActiveX interface class, in the Class Workshop, click the View > Inherited command.

Figure 4-1 fdir Class Methods, Attributes, and Events



If these classes are already available in your development repository, you can skip this section and move on to "Developing an iPlanet UDS Application that Uses ActiveX Controls" on page 109.

To generate these TOOL classes, use the `Olegen` utility against the type library for the control to generate a .pex file. This .pex file defines a project containing TOOL classes, methods, attributes, and events that map to those defined for the control. You need to import this .pex and include the project as a supplier to your iPlanet UDS application.

➤ **To produce TOOL classes for a control**

1. Install the ActiveX controls on your system.
2. Run the `Olegen` utility. This utility generates a .pex file.
3. Import the .pex file that you generated with the `Olegen` utility into your development repository.

These steps are described in greater detail in the following sections.

Install the ActiveX Control on Your System

When you install an ActiveX control on your system, the installation program registers the control in the Windows registry. When iPlanet UDS accesses the properties and methods of this control, iPlanet UDS checks the registry to locate the control on your system.

If you choose to run the `Olegen` utility against the type library explicitly using the **-it** flag, you need to know where the file for the control with the extension .ocx resides on your system. The `Olegen` utility is discussed in "Run the Olegen Utility."

Run the Olegen Utility

iPlanet UDS provides an `Olegen` utility that generates a TOOL project definition and classes using the type library provided by the ActiveX control.

To run the `Olegen` utility, you must be running under Windows NT or Windows 95. You can start this utility using the Windows Run dialog.

The `olegen.exe` file is in the `FORTE_ROOT\install\bin` directory.

The syntax for starting the `Olegen` utility is:

olegen *input_specification* **[-of** *output_file_name*] **[-ai]**

input_specification is one of the following:

input_specification	Description
-it *type_library*	*type_library* is the file name of the type library for the ActiveX control. The type libraries for ActiveX controls have the extension .ocx. A type library might contain information for more than one control. If the type library is in a directory different from the current directory, you need to specify its path.
-ip *ProgID*	*ProgID* is the programmatic identifier of ActiveX control. These identifiers typically have the syntax *application_name***.***object*, for example, "excel.application." You can locate the programmatic identifiers of ActiveX controls by using the registry editor and looking in the HKEY_CLASSES_ROOT on the Local Machine window. The documentation for an ActiveX control should provide its programmatic identifier.
-ic *CLSID*	*CLSID* is the unique identifier string for an ActiveX control. The CLSID is a string of 32 hex digits enclosed in braces, for example: {00021A00-0000-0000-C000-000000000135}.

The **-of** flag, which is optional, specifies the path and file name for the generated .pex file. The default file name is `olegen.pex`, and the default path is the current working directory.

The **-ai** flag ("assume input"), which is optional, specifies how the `Olegen` utility interprets method parameters that do not specify a passing mode. By default, `Olegen` interprets all method parameters that do not specify a passing mode as input Variant objects. When the **-ai** flag is specified, `Olegen` interprets all parameters that do not specify a passing mode as input parameters of an iPlanet UDS data type.

Because it is usually easier to work with iPlanet UDS data types than with Variant objects, we recommend that you specify the **-ai** flag. The default is not to assume that parameters are input parameters to be compatible with earlier releases of iPlanet UDS.

Variant classes are OLE-specific classes that require special handling to convert their data into standard TOOL classes. For more information about using Variant objects, see "Converting Data to a Variant Object" on page 57 and "Converting a Variant Object to a TOOL Object or Data Type" on page 58.

For example, to generate a project definition that maps to the iPlanet UDS example fdir32 ActiveX control, you can start the `Olegen` utility using the following command:

```
olegen -it c:\controls\fdir32.ocx
    -of fdir32.pex -ai
```

For information about the type library name, the programmatic ID, or the CLSID for a given control, see the documentation for the specific ActiveX control.

For information about how the `Olegen` utility generates the TOOL classes, see Appendix B, "Olegen Mapping Conventions."

Import the Generated Project Definition .pex File

Import the .pex file generated by the `Olegen` utility, using either the Import command in the Repository Workshop, or the `ImportPlan` command in Fscript. For specific instructions for importing a project definition, see *A Guide to the iPlanet UDS Workshops* and the *Fscript Reference Guide*.

Developing an iPlanet UDS Application that Uses ActiveX Controls

This section explains how to write an iPlanet UDS application that uses the default dispatch interface and events of an ActiveX custom control. If you don't know whether you are using the default dispatch interface and events or not, you probably are.

If you want to use one of the other dispatch interfaces, see iPlanet UDS Technote 10825.

If you only want to use the ActiveX control as a display mechanism, or only call a few functions on the control, but not catch any events on the control, you can follow the simpler steps described in "Using ActiveX Controls to Display Information" on page 105.

Before You Start

Before you can include an ActiveX control in your application, you need to have the ActiveX control installed on your system.

When you install an ActiveX control on your system, the installation program registers the control in the Windows registry. When iPlanet UDS accesses the properties and methods of this control, iPlanet UDS checks the registry to locate the control on your system. The ActiveX control file that you need to find is the file with the extension .ocx. This file contains the type library for the control. If you choose to run the `olegen` utility against the type library explicitly using the **-it** flag, you need to know where this file resides on your system.

Before you can program your application to catch any ActiveX control events in your TOOL application, you need to have TOOL classes that correspond to that control in your development repository. If you do not yet have these TOOL classes available, follow the steps described in "Producing TOOL Classes For an ActiveX Control" on page 106.

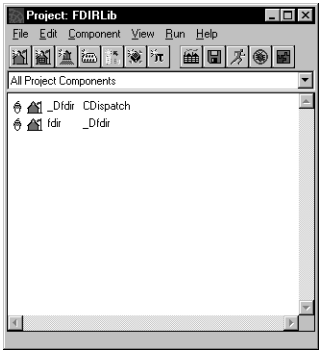
A project that contains classes for an ActiveX control contains the following kinds of classes:

dispatch interface class A subclass of the `CDispatch` class of the OLE library. This class provides the methods and attributes for the ActiveX control. To interact with a control, create an instance of the ActiveX interface class.

ActiveX interface class A subclass of the default dispatch interface class for the control. The ActiveX interface class inherits the attributes and methods of the default dispatch interface. In addition, this class defines the events that can be sent by the control and the methods that handle these events. To see all the methods, attributes, and events available in the ActiveX interface class, in the Class Workshop, click the View > Inherited command.

The following figure shows the contents of a project generated by the `Olegen` utility. _Dfdir is the dispatch interface class and fdir is the ActiveX interface class.

Figure 4-2 FDIRLib project generated for the FourDir ActiveX control



Restrictions

If you print a window containing an ActiveX control using the PrintDocument class, the ActiveXField widget will be printed, but not the control itself. However, you can still take a snapshot of the iPlanet UDS window, and the ActiveX control will be printed with the rest of the window.

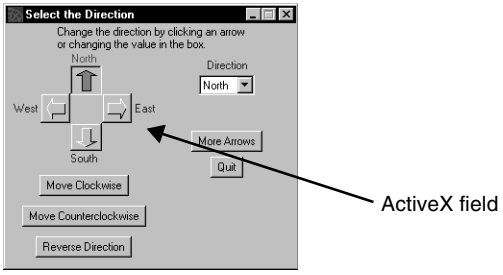
If you define a window that contains an ActiveXField widget that maps to an ActiveX control, and you want to subclass this window, you should define the ActiveX control that maps to the ActiveXField widget dynamically, as described in "In TOOL Code—Dynamic Definition" on page 117. Information about the ActiveX controls that are inserted statically, as described in "In the Window Workshop—Static Definition" on page 114 is not inherited.

In general, you should only develop applications that use ActiveX controls on platforms that support the controls (Microsoft Windows NT and Windows 95). You need to have the ActiveX control installed and registered in Windows before you can statically define an ActiveX control using the Window Workshop, as described in "In the Window Workshop—Static Definition" on page 114. You can write TOOL code that dynamically defines ActiveX fields and ActiveX controls on other platforms; however, you need to test the application on a platform that supports the ActiveX control.

Overview

When you use an ActiveX control in your iPlanet UDS window, you need to use an ActiveXField widget to contain the ActiveX control. The ActiveX control is actually a mapped type of the ActiveXField widget.

Figure 4-3 ActiveX Field Containing an ActiveX control



When you use an ActiveX control in your iPlanet UDS application, you need to perform the following steps:

1. In projects that use the ActiveX control, specify the project for the ActiveX control interface as a supplier plan.
2. In the Window Workshop or in your TOOL code, define an ActiveXField widget and its ActiveX control.
3. Invoke methods and access properties of the control using methods and attributes of the ActiveX interface class.
4. In your TOOL code, handle events sent by the ActiveX control, for example, Click or MouseMove.

These steps are explained in detail in the following sections.

Specify the Supplier Plans

In projects that use the ActiveX control, specify the project for the ActiveX control interface as a supplier plan. For information about specifying supplier plans, see *A Guide to the iPlanet UDS Workshops*.

Define an ActiveXField Widget

This section describes how to do each step using the Window Workshop, and then how to do similar tasks in TOOL code.

NOTE You can define ActiveX fields on any platform, however, you can only insert ActiveX controls, view their properties, and run applications that use ActiveX controls on Windows machines that have the required ActiveX controls installed.

In general, to define an ActiveX control, you need to create an ActiveX field and specify the type of control you are putting into the ActiveX field.

By default, the mapped type for the ActiveXField widget is CDispatch. Because all the ActiveX interface classes are subclasses of `CDispatch`, you could leave the mapped type CDispatch and cast the mapped type, as appropriate. This approach can be useful if you plan to use different ActiveX controls in your ActiveXField widget. If you choose to leave the mapped type CDispatch, you need to make the OLE library a supplier project for your project.

NOTE You should specify a specific ActiveX interface class as the mapped type for the ActiveXField widget, unless you plan to insert different ActiveX controls in the ActiveXField widget dynamically. Specifying a specific class lets the compiler check data types and produces useful runtime errors, if necessary.

In general, you should only develop applications that use ActiveX controls on platforms that support the controls (Microsoft Windows NT and Windows 95). You need to have the ActiveX control installed and registered in Windows before you can statically define an ActiveX control using the Window Workshop, as described in "In the Window Workshop—Static Definition" on page 114. You can write TOOL code that dynamically defines ActiveX fields and ActiveX controls on other platforms; however, you need to test the application on a platform that supports the ActiveX control.

In the Window Workshop—Static Definition

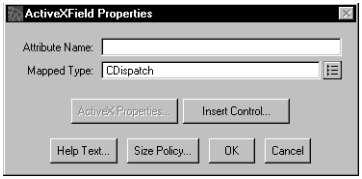
In the Window Workshop, you can define an ActiveXField widget, select the type for the mapped type, insert the ActiveX control, and set the initial properties for the ActiveX control.

When you use an ActiveX control in your iPlanet UDS window, you need to use an ActiveX field widget to contain the ActiveX control. The ActiveX control is actually a mapped attribute of the ActiveX field. To create the ActiveX field, you use the Widget > New > ActiveXField command.

The following table describes the properties on the ActiveXField Properties dialog, shown in the figure below:

Use This Property	For This Purpose
Attribute Name	Sets an attribute name for the picture field.
Mapped Type	Specifies the mapped data type for the OLE field. This value must be `CDispatch` or a subclass of `CDispatch`.
ActiveX Properties button	Lets you view and set properties of the ActiveX control. This button displays a dialog containing the properties defined by the ActiveX control.
Insert Control button	Lets you add an ActiveX control. This button displays a dialog, in which you can define the type of ActiveX control.
Help Text	Opens the Help Text dialog for the field.
Size Policy	Opens the SIze Policy dialog for the field.

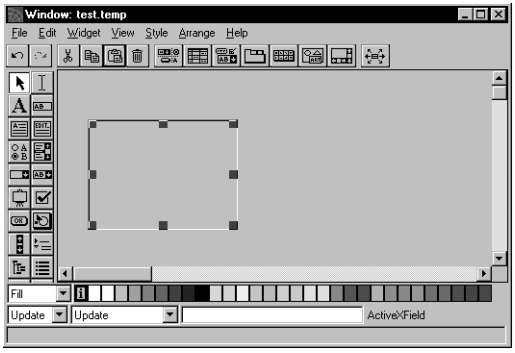
Figure 4-4 ActiveXField Properties dialog



➤ **To define an ActiveXField widget**

1. Choose the Widget > New > ActiveXField command.
2. Draw an ActiveX field of the size you want in the window.

Figure 4-5 A new ActiveXField widget



➤ **To define the mapped type**

1. Open the ActiveXField Properties dialog by double-clicking on the ActiveXField widget.
2. In the Mapped Type field, enter the name of the ActiveX interface class, or use the browser button to display a list of available classes, and select the ActiveX interface class.

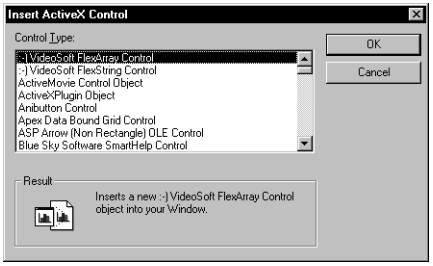
   For example, if you want to set the mapped type to the class for the FourDir control, set the Mapped Type field to the FDIRLib.fdir class.

NOTE If the mapped type is not CDispatch, the specified type must match whatever control is inserted; otherwise, you will get a runtime error.

➤ **To insert the ActiveX control into the ActiveX field**

1. In the ActiveXField Properties dialog, click the Insert Control button.
2. In the Insert Control dialog, select the ActiveX control that you want to insert into the ActiveX field.

Figure 4-6 Insert Control dialog

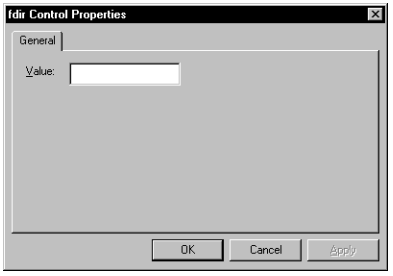


NOTE You can only insert ActiveX controls into ActiveX fields on Windows platforms where the ActiveX controls have been installed and registered.

➤ **To set the initial property values for the ActiveX control**

1. In the ActiveXField Properties dialog, click the ActiveX Properties button.
2. In the tab folders, set the values that are appropriate for the ActiveX control. For specific information about these properties, see the documentation for the ActiveX control.

Figure 4-7 ActiveX Control Properties



NOTE When you change values in this dialog and click either OK or Apply, the values are changed for the ActiveX control, even if you later cancel out of the ActiveXField Properties dialog. If you click the Cancel button on this dialog, any changed values are not changed for the ActiveX control.

In TOOL Code—Dynamic Definition

In TOOL code, you can define an ActiveXField widget and associate it with a particular ActiveX control with the ActiveX field in the TOOL code.

➤ **To define an ActiveXField widget in your TOOL code**

Put an ActiveXField widget in the window, but leave the Mapped Type field as CDispatch.

➤ **To insert an ActiveX control into the ActiveX field**

1. Create an object of the ActiveX interface class, using code like the following:

   ```
   dynamicArrows : fdirSub = new();
   ```

   In this example, fdirSub is a subclass of FDIRLib.fdir, which is the ActiveX interface class for the FourDir control.

2. Insert the ActiveX control into the ActiveX field by setting OleObjectValue attribute of the ActiveXField widget with the new instance of the ActiveX interface class, as shown:

   ```
   self.<DynamicACField>.OleObjectValue = dynamicArrows;
   ```

When you define the ActiveX control dynamically, you can later place another control in the ActiveX field.

Invoke Methods and Access Properties of the Control

In your TOOL code, invoke methods and access properties of the control using methods and attributes of the ActiveX interface class. To see all the methods, attributes, and events available in the ActiveX interface class, in the Class Workshop, click the View > Inherited command.

You should also have whatever documentation is available for the control you are using so that you can determine what the methods, properties, and events are intended to do or mean.

You can only invoke methods and access properties of the ActiveX control after the window containing the ActiveXField widget has been opened by invoking the `Open()` method. You can instantiate the ActiveX interface class using the new() function, but the ActiveX control itself does not exist until the window has been opened.

In some cases, `Olegen` cannot determine the data type of a property, a parameter, or a return value. In these cases, you need to use Variant objects to pass the data to and retrieve data from the ActiveX control. For information about using Variant objects, see "Dealing with Variant Objects" on page 56.

NOTE When you import the .pex file generated by the `Olegen` utility, iPlanet UDS removes the quotation marks from the methods. However, when you use a method whose name or whose parameters' names are TOOL reserved words, then you need to specify double quotation marks around the names that are reserved words. To see the list of TOOL reserved words, see *TOOL Reference Guide*.

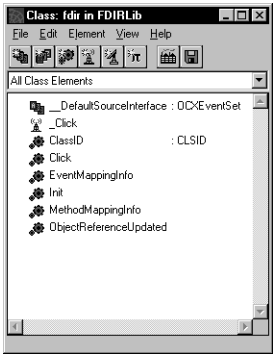
Handle Events Posted by the ActiveX Control

In your TOOL code, you can handle events sent by the ActiveX control, for example, Click or Change.

When the `Olegen` utility generates an ActiveX interface class for an ActiveX control, the utility generates a set of methods that are automatically invoked when the ActiveX control sends an event. These methods have the same names as the events that can be sent by the control. One of these methods is automatically invoked synchronously whenever the ActiveX control posts the corresponding event.

The following figure shows the methods, attributes, and events defined by the ActiveX interface class for the FourDir ActiveX control. Click is the generated method that is automatically called when the control posts an event, and _Click is the asynchronous event that the default generated method automatically posts.

Figure 4-8 fdir Class Methods, Attributes, and Events



You can also see that a set of iPlanet UDS events have been generated. The iPlanet UDS event names have the same names as the control's events, except that they start with an underscore character (_). For example, if the control can send the Click event, the `Olegen` utility generates an iPlanet UDS event called _Click.

For example, the FourDir custom control defines an event called Click. This event maps to the _Click event and the Click method in the ActiveX interface class.

Asynchronous events In an iPlanet UDS window, ActiveX controls behave, by default, like iPlanet UDS widgets. When a control senses a user action or change in condition, iPlanet UDS posts an asynchronous event on the control's ActiveX interface class. The iPlanet UDS events defined for each control have the same name as the control's events, with a preceding underscore character (_). For example, if the ActiveX control has an event called Click, the default event that is posted is _Click. In this case, you can handle the event using an event loop, just as you would for any other iPlanet UDS event. Asynchronous events are available for Windows 95 and Windows NT.

The following example from an event loop shows how an iPlanet UDS application could catch a _Click event posted by the FourDir ActiveX control:

```
when self.DirectionArrows._Click do
  self.CurrentDirection = self.DirectionArrows.Value;
  self.SetList(dir = self.CurrentDirection);
  self.window.UpdateDisplay();
```

Project: ActiveXDemo • **Class:** ActiveXWin • **Method:** Display

Synchronous events However, if you want to have your application respond differently to an ActiveX control event, you can override the default method generated for the event. Because the OLE event mechanism calls TOOL methods directly to send a control's events, you can also process the control's events synchronously.

To process events synchronously, subclass the ActiveX interface class to override the default methods to define some processing that will occur when the control sends its events. You can override default event methods for applications running on Windows 95 and Windows NT.

➤ **To override the default method generated for an ActiveX event**

1. Subclass the ActiveX interface class for the ActiveX control.

   Although iPlanet UDS does not prevent you from directly editing the methods of the generated ActiveX interface class, you should change the methods by overriding them in a subclass to avoid maintenance and support problems.

2. Define a method in this subclass that overrides the default generated method in the ActiveX interface class.

NOTE	Be careful that the method does not include a time-consuming activity that might make the control appear extremely slow to the user.

The following example shows how you can subclass the ActiveX interface for the FourDir control (FDIRLib.fdir) and override the default Click method to select each arrow before finally selecting the arrow that was clicked:

```
-- This method moves the selected arrow around the FourDir
-- control clockwise.
task.Part.LogMgr.putLine(source='Entered fdirSub.Click.');
-- Define a timer so that the arrow selection moves slowly
-- enough to be seen.
myTimer : Timer = new();
myTimer.TickInterval = 200;
myTimer.IsActive = True;
for x in 1 to 4 do
  event loop
    when myTimer.Tick do
      self.MoveClockWise();
      exit;
```

Project: ActiveXDemo • **Class:** fdirSub • **Method:** Click

```
  end event;
end for;
task.Part.LogMgr.putLine(source='Exiting fdirSub.Click.');
```

Project: ActiveXDemo • **Class:** fdirSub • **Method:** Click

In this example, the Click event is not posted.

Partitioning the TOOL Application

TOOL classes that subclass or instantiate ActiveX interface classes can only run in client partitions that have Windows NT or Windows 95 installed.

You should configure your application using the Partition Workshop or Fscript, as described in *A Guide to the iPlanet UDS Workshops* or the *Fscript Reference Guide*.

When you configure your application, iPlanet UDS, by default, puts the client partitions containing ActiveX controls on all available client nodes. You need to remove the client partitions that use ActiveX controls from nodes that run non-Windows platforms.

Making the Distribution and Installing the Application

You can make the distribution just as you would for any regular TOOL application. You can either make the distribution from the Partition Workshop or from within Fscript. For more information about using the Partition Workshop or Fscript, see *A Guide to the iPlanet UDS Workshops* or the *Fscript Reference Guide*.

Install ActiveX Controls Where Client Partitions are Installed

You can install the client application just as you would any other TOOL client application, except that you must ensure that the ActiveX controls used in the application are installed on the nodes where you are installing the client partitions that use the ActiveX controls.

Assuming that you have taken care of any licensing issues with the ActiveX controls, you can minimize the work required for installing the controls as part of the iPlanet UDS application.

To have the installation files for the ActiveX controls automatically downloaded with the client partition files, place the files for the control in the following application distribution directory path:

FORTE_ROOT/appdist/*environment_ID/distribution_ID*/cl#/*platform_ID*/*component_ID*

platform_ID is PC_NT for Windows 95 and Windows NT.

component_ID is the *partition_ID* for the client partition (ending in 0).

Run the ActiveX installation program After the files for the ActiveX control and the client partition are on the client nodes, run the installation program for the ActiveX control, which places the files in the appropriate place on the system and registers the ActiveX control in the Windows Registry.

All the downloaded files for the application are placed in the following directory on the client node:

FORTE_ROOT\userapp*distribution_id***\cl#**

For detailed information about installing iPlanet UDS applications, see *iPlanet UDS System Management Guide*.

Troubleshooting

If iPlanet UDS cannot locate the control at runtime, you should make sure that:

- the control is registered in the Windows registry
- the control has not been deleted
- the registry entry for the control is up-to-date, especially if you have deleted and reinstalled the control

If the control is correctly registered, you should also check that the control works in a Microsoft application like Microsoft Visual Basic. If the control does not work in Microsoft Visual Basic, it will probably also not work in iPlanet UDS.

Chapter 5

Using Dynamic Data Exchange

This chapter discusses how to enable iPlanet UDS applications to communicate with Windows applications using Dynamic Data Exchange (DDE). For details on the class reference for the classes in the DDEProject library see the iPlanet UDS online Help.

Topics covered in this chapter include:

- establishing an iPlanet UDS application as a DDE client or DDE server
- defining applications and topics
- initiating a conversation
- executing a command
- setting a link
- terminating a session

This information is contained in the individual class and method reference sections.

For more information specific to DDE, refer to Microsoft's documentation.

About Dynamic Data Exchange

Dynamic Data Exchange is a mechanism for interprocess communication supported in Windows applications.

Clients and servers DDE is based on the messaging system built into Windows. Two Window programs can carry on a DDE *conversation* by posting messages to each other. These two programs are known as the *server* and the *client*. A DDE server is the program that has access to data that might be useful to other programs. A DDE client is the program that obtains this data from the server.

A single application can be both a client to one program and a server to another, but this situation requires two different DDE conversations. A server can deliver data to multiple clients, and a client can access data from multiple servers, again using multiple conversations.

The programs involved in a DDE conversation need not be specifically coded to work with each other. A writer of a DDE server publicly documents how the data is identified. A user of a program acting as a DDE client can use this information to establish a DDE conversation between the two programs.

iPlanet UDS Integration with DDE

iPlanet UDS provides a set of classes that let iPlanet UDS applications participate in DDE conversations with Windows products. An iPlanet UDS application can be a client or a server. For example, an iPlanet UDS inventory application (client) might need data that is tracked in an Excel spreadsheet (server). With DDE, you can establish links to the spreadsheet which update the inventory application with the current data each time the spreadsheet is changed. Likewise, a Microsoft Word user might need database information that is managed by an iPlanet UDS server application. Using DDE, data can be extracted from the server directly into the word document.

iPlanet UDS's DDE Classes

The iPlanet UDS library named DDEProject provides the following classes to implement the iPlanet UDS/DDE integration:

iPlanet UDS Class	Description
DDEClient	Holds the name of the client application for identification purposes if more than one conversation is established with the same server. A DDEClient is created each time a client application initiates a conversation with an iPlanet UDS server. The DDEClient object simply holds the name of the client application for identification purposes if more than one conversation is established with the same server.
DDEConversation	Used by an iPlanet UDS client application to establish the connection to a DDE server. All requests for data are invoked by this object.
DDEObject	Abstract superclass to all other DDE classes.
DDEServer	Used by an iPlanet UDS server application to respond to requests from a DDE client application.

Using Methods and Events

The classes you will use most often in your DDE integration are DDEConversation and DDEServer objects. Each of the DDE classes has a set of methods and events. You must understand the relationship between these methods and events in the context of a server application and a client application before you can effectively use these classes.

iPlanet UDS as a DDE Client When an iPlanet UDS application is the DDE client, it initiates requests. The requests are initiated through the DDEConversation methods. Most of the DDEConversation methods use return events to signal the completion of a request and to pass back data from the server.

iPlanet UDS as a DDE Server When an iPlanet UDS application is the DDE server, this application responds to requests for data from a client DDE application. Therefore, this DDE server application responds to posted events by invoking a method to fulfill the request.

The table below displays the paired methods and events of the DDEConversation and DDEServer classes. Note that DDEConversation methods post events, while DDEServer events initiate methods:

DDEConversation		DDEServer	
method	**event**	**event**	**method**
InitiateConnection	none	CommandRequest	CommandResponse
RequestExec	ExecComplete	Connected	StartServer
RequestGetItem	GetItemComplete	Disconnected	EndServer
RequestLinkEnd	LinkStatus	GetItemRequest	GetItemResponse
RequestLinkStart	LinkStatus	LinkedItemRequest	UpdateItem
RequestPutItem	PutItemComplete	LinkEndRequest	none
TerminateConnection	Disconnected	LinkStartRequest	LinkStartResponse
		PutItemRequest	PutItemResponse

Part 2

Creating an XML Server for UDS Applications

Part 2 of *Integrating with External Systems* provides usage and reference information about exporting iPlanet UDS service objects as XML servers. It also contains information about writing client applications that access the XML server.

Part 2 contains the following chapters:

Chapter 6

Overview of XML Server

This chapter provides an overview of iPlanet UDS XML servers, including information on how you can create XML servers that can be accessed by external client applications.

About UDS XML Servers

iPlanet UDS allows you to export a UDS service object as an XML server, which can then be subsequently accessed by external client applications. Any application that can send XML messages over HTTP/HTTPS and has access to a SOAP library can interact directly as a client to the XML server. iPlanet UDS facilitates the creation of SOAP client applications in two ways:

- A UDS XML server can generate the Java source files that can be used when creating SOAP client applications
- A UDS XML server automatically generates a WSDL file that can be used by third-party utilities that automatically generate client applications or that publish services using a third-party registry

About SOAP

iPlanet UDS XML Services is an implementation of the Simple Object Access Protocol (SOAP) specification, which is an XML-based protocol for the exchange of information in a decentralized, distributed environment.

SOAP messages are XML documents that are typically exchanged over the HTTP/HTTPS protocols. SOAP messages consist of a SOAP envelope (which defines the framework for the contents of a message), SOAP encoding rules (which define how instances of objects are serialized and marshalled between applications), and the SOAP RPC representation (which defines the conventions for remote procedure calls and responses).

A UDS XML server implements the message exchange between the server and the client application, shielding you from the task of creating, parsing, and responding to XML documents embedded in SOAP messages. Files generated by a UDS XML server facilitate the creation of client applications that communicate with the server.

For more information about the SOAP protocol, refer to http://www.w3.org/TR/SOAP. For information on the version of SOAP implemented by iPlanet UDS, refer to the iPlanet UDS Platform Matrix at http://www.forte.com/support/platforms.html.

UDS XML Services Architecture

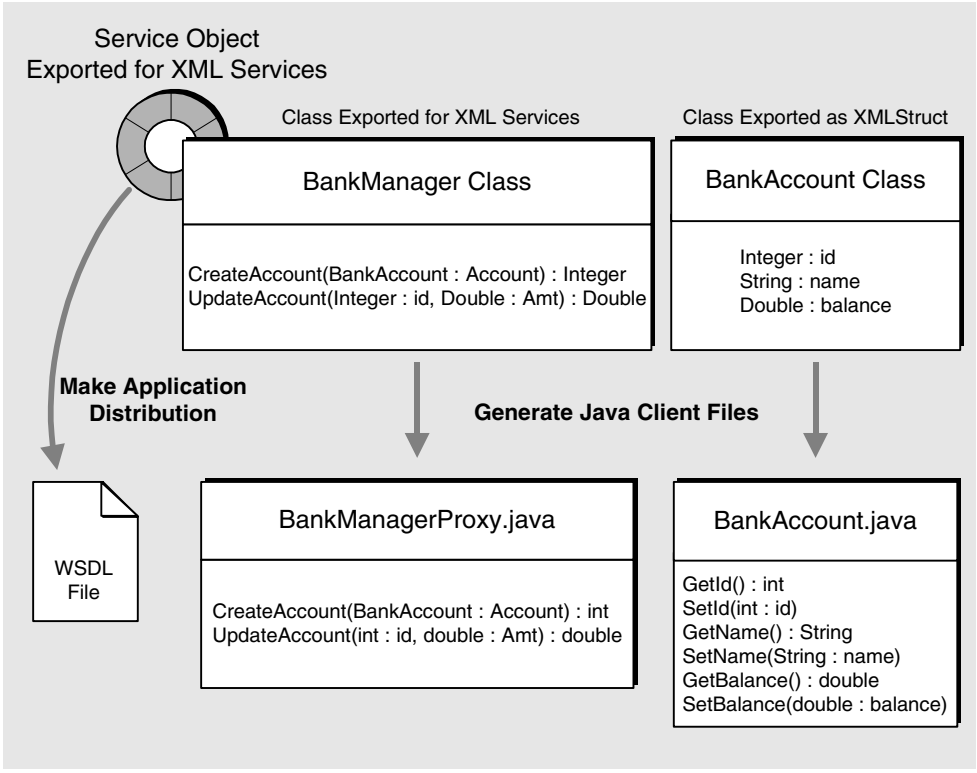
This section illustrates how an iPlanet UDS service object is exported as an XML server. The example in this section is based on the XML server example provided with your distribution at:

```
FORTE_ROOT/install/examples/extsys/xmlsvr
```

This example implements a basic banking service that allows a client to create bank accounts, display account information, and make deposits and withdrawals. The BankManager class is implemented as a service object with methods for accessing the banking services. A bank account object is represented as a SOAP *compound datatype* that must be serialized before transporting it in SOAP messages.

Figure 6-1 illustrates how the BankManager class is exported for XML services. It shows two service object methods for creating and updating bank accounts. A bank account is marked as an *XMLStruct* in the UDS application so it can be exported as a SOAP compound datatype.

Figure 6-1 Service Object Exported for XML Services

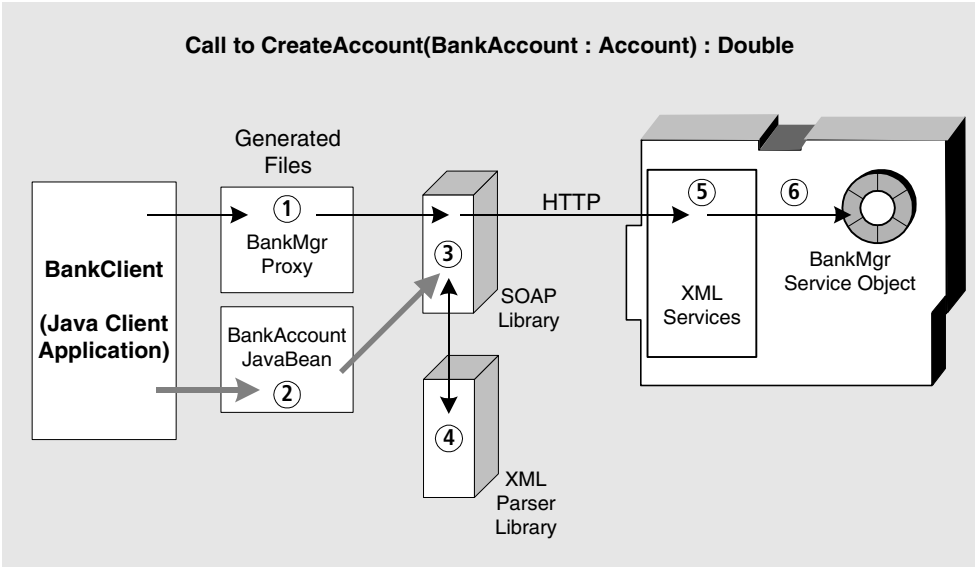


Generated WSDL Files The Web Services Description Language (WSDL) is a specification that uses XML documents to describe Web services. A WSDL file typically encodes network service information such as ports, messages, operations, and binding. When you make an application distribution of an XML server, iPlanet UDS automatically generates a WSDL file that describes the XML server. You typically use this file as input to third-party utilities to generate client applications to the XML server or to publish the exported services on a third-party registry. (For more information on the WSDL specification, refer to http://www.w3.org/TR/wsdl.)

Generated Java Files After creating an XML server application, you can issue the GenerateXMLServerJava Fscript command to generate Java source files that can be used by Java client applications to access the exported services. This command also generates JavaBean source files (as needed) that are used by the Java client to serialize SOAP compound datatypes (such as the bank account object) for transport in SOAP messages.

Figure 6-2 illustrates how a Java client application can use the generated Java files to access the exported bank services. This figure illustrates a call from the client to CreateAccount on the XML server. This call passes a bank account object, which must be serialized for SOAP transport.

Figure 6-2 Accessing XML Services from a Java Client Application



The call to CreateAccount proceeds as follows:

1. The Java client calls the BankManagerProxy method that corresponds to the CreateAccount method exported by the server.

   This method requires a bank account object as a parameter.

2. When building the call to CreateAccount, the Java client instantiates a JavaBean for the bank account, which the SOAP library uses to serialize the object.

3. The BankManagerProxy accesses a SOAP library to embed the call in a SOAP message.

4. The SOAP library accesses an XML parser library to encode the call in a SOAP XML document.

5. The SOAP library sends the message over HTTP to the XML services implemented by the BankManager service object.
6. XML services decodes the SOAP message and passes the request to the BankManager service object, which creates the requested account.

XML Services Environment

No special environment setup is required to export a service object as an XML server. However, the environment for client applications must be properly configured to access a SOAP implementation. For Java clients, environment considerations include the following:

- Java Development Kit (JDK) version
- SOAP implementation version
- XML parser
- CLASSPATH properly configured to access SOAP and XML parser libraries

For information on specific requirements for client access to iPlanet UDS XML services, refer to the Platform Matrix, available at http://www.forte.com/support/platforms.html.

Creating an XML Server and Client Applications

You can create an XML server using options available from the Repository Workshop or you can create the server using Fscript commands. This section provides a high-level view of the steps necessary to create an XML server and corresponding client applications. Detailed steps are provided in subsequent chapters.

Basic Steps to Create an XML Server

The following procedure describes the basic steps necessary to export an iPlanet UDS service object as an XML server. Refer to Chapter 7, "Exporting an iPlanet UDS Service Object as an XML Server" for details on this procedure.

➤ **To export an iPlanet UDS service object as an XML server**

1. For the service object classes that are being exported for XML services, mark any methods you do not want to export.

   By default, public methods are marked for export.

2. Mark complex datatypes for export that are either passed as method parameters or returned as values.

3. Mark any attributes you do not want to export.

   By default, public attributes are marked for export.

4. Export and distribute the XML server.

   When exporting the XML server, specify any port and security information.

5. If you plan to access the XML server from a Java client, generate a Java proxy and any corresponding Java files that may be needed.

6. Start the XML server.

Basic Steps to Create a Java Client Application

The following procedure describes the basic steps necessary to create a Java client application for an XML server. Refer to Chapter 8, "Creating Java Client Applications for an XML Server" for details on this procedure.

➤ **To create a Java client application to an XML server**

1. Be sure your SOAP environment is correctly set up, as described in "XML Services Environment" on page 133.

2. In your Java client application, import the Java files generated from the previous procedure.

3. In your Java client, use the methods in the generated Java proxy to access the service object.

   To pass SOAP compound datatypes to or from the server, use the generated JavaBean file.

Terminology

Compound Datatype In SOAP encoding style, a compound type is a composite type, with each part being either a SOAP basic type or another compound type ultimately derived from SOAP basic types. SOAP uses compound datatypes to serialize objects passed in SOAP messages.

Simple Datatype In SOAP encoding style, a simple type is scalar datatype.

SOAP The Simple Object Access Protocol specifies a way to exchange information in a decentralized, distributed environment. SOAP encodes messages as XML documents containing three parts: an envelope providing information about the message and how to process it, a set of encoding rules that defines objects included in the message, and a convention for representing remote procedure calls and responses. SOAP messages are typically sent using the HTTP/HTTPS protocols. iPlanet UDS XML services handles the creation, sending, and decoding of SOAP messages for XML server applications. For more information on SOAP, refer to http://www.w3.org/TR/SOAP.

SOAP Fault A SOAP Fault is an element of a SOAP message that contains error and/or status information. If present, the SOAP Fault appears within the body of a SOAP message.

WSDL The Web Services Description Language is a specification that uses XML documents to describe Web services. A WSDL file typically encodes network service information such as ports, messages, operations, and binding. iPlanet UDS XML server application automatically generates a WSDL file that describes the XML server. You typically use this file as input to third-party utilities to generate client applications to the XML server or to publish the exported services on an external registry. For more information on the WSDL specification, refer to http://www.w3.org/TR/wsdl.

XML Server An iPlanet UDS application with one or more service objects exported to external systems using XML services. Typically, Java client applications access XML servers, but any external system capable of SOAP messaging can also access an XML server.

XML Services Access to the methods of an iPlanet UDS service object using XML messaging available from SOAP. XML services allow external systems to UDS to communicate with UDS applications.

XMLStruct In an XML server application, an XMLStruct encodes iPlanet UDS datatypes as SOAP compound datatypes for transport in SOAP messages. XMLStructs allow the server to expose user-defined objects, which are ultimately composed of UDS scalar types, as method parameters or return values. You mark such UDS objects as XMLStructs in the UDS Project Workshop.

Chapter 7

Exporting an iPlanet UDS Service Object as an XML Server

This chapter describes how to export an iPlanet UDS service object as an XML server. Any iPlanet UDS service object that is environment visible can be exported as an XML Server. Because of the nature of representing data across different architectures, and because of considerations of translating datatypes from one system to another, service objects created for export may require special design considerations.

For information on creating and using service objects, refer to the *iPlanet UDS Programming Guide*.

Exporting a service object as an XML server involves the following steps:

- Defining the methods to export
- Specifying the data to export
- Exporting the service object using either the Project Workshop or Fscript commands.

Exporting Service Object Methods

When you export a service object as an XML server, all of its public methods, with some restrictions, are callable from client applications. Keep in mind the following restrictions on methods you export in an XML service object:

- The Init method is not exported.

  Although the Init method is not exported, it is still called by the server when it instantiates the object.

- Methods marked as private are not exported.

- Methods containing compound datatypes as a parameter or return value are only exported under certain conditions.

  Compound data types in SOAP are objects that are treated as arrays or C-like structures of simple data types. Most compound datatypes can be marked for export. Refer to "SOAP Compound Data Types" on page 141 for more information.

Exporting Service Object Data

The SOAP specification defines parameter data as either a simple type or a compound type. *Simple types* represent basic data types such as integers, strings, and floating point values. Compound types represent arrays and structures composed of basic data types and/or other structures.

SOAP Simple Data Types

An XML server attempts to represent most basic UDS datatypes as SOAP simple datatypes. Because programming languages vary in how they handle basic datatypes, and because some UDS classes provide functionality missing from simple datatypes, you should be careful in how you define the data structures in both your server and client applications. In many cases, you may want to write wrapper classes on either the client or server to handle datatypes correctly on the target platforms.

UDS Integer Data Types

Table 7-1 lists UDS framework integer data types and the corresponding conversions to Java datatypes made by UDS XML services.

Table 7-1 Conversions for UDS Integer Data Types

UDS Framework Data Type	Translation to Java	XML Type	Portable
int	short	xsd:short	no
short	short	xsd:short	no
long	long	xsd:int	no
i1	byte	xsd:byte	yes
i2	short	xsd:short	yes
i4	int	xsd:int	yes
Integer	int	xsd:int	yes
ui1	short	xsd:unsignedByte	yes[a]
ui2	int	xsd:unsignedShort	yes[a]
ui4	long	xsd:unsignedInt	yes[a]

a. Unsigned integers are not portable to Java types

NOTE The UDS data types listed as portable are portable across different architectures. However, UDS XML services cannot guarantee that SOAP implementations handle translations according to XML schema requirements. For example, you may have to use custom serializers to handle unsigned integers.

UDS Boolean and String Data Types

The iPlanet UDS Framework library supports scalar types for Boolean and String values, which are portable across platforms.

Table 7-2 lists UDS framework Boolean and String data types, and the corresponding conversions to Java datatypes made by UDS XML services.

Table 7-2 Conversions for UDS Boolean and String Data Types

UDS Framework Data Type	Translation to Java	XML Type
Boolean	boolean	xsd:boolean
String	String	xsd:string

UDS Float Data Types

The iPlanet UDS Framework library supports two float data types. The exact precision of a float data type is dependent on machine architecture. The portability of float data types depends not only on the machine architecture of the server and clients, but also on the precision supported by programming language of the client.

Table 7-3 lists UDS framework float data types and the corresponding conversions to Java datatypes made by UDS XML services.

Table 7-3 Conversions for UDS Float Data Types

UDS Framework Data Type	Translation to Java	XML Type
Float	float	xsd:float
Double	double	xsd:double

UDS DataValue Classes

An XML server supports conversions of some UDS DataValue subclasses to simple SOAP datatypes. Table 7-4 lists the conversions to Java data types for DataValue subclasses.

NOTE If a parameter or return value to a method is a DataValue subclass that is not supported for conversion, then the method is not exported by XML services.

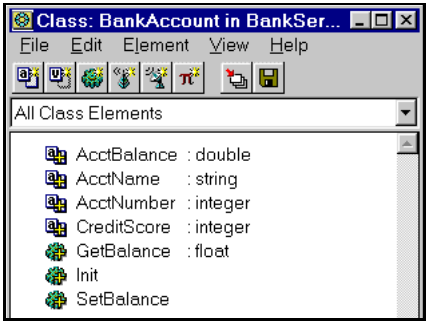
Table 7-4 Conversions for UDS DataValue Classes

UDS Framework DataValue Class	Translation to Java	XML Type
BooleanData	boolean	xsd:boolean
DoubleData	double	xsd:double
FloatData	float	xsd:float
IntegerData	int	xsd:int
TextData	String	xsd:string

SOAP Compound Data Types

Objects created from user-defined UDS classes and arrays are considered to be SOAP compound data types. Such classes must be marked as an XMLStruct before they can be exported.

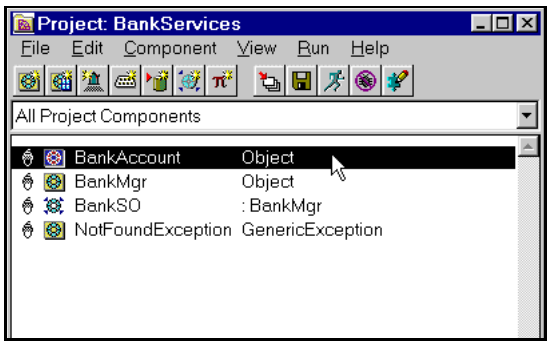
For example, suppose your application defines a BankAccount object that contains a string attribute representing the account name, an integer attribute representing the account number, and a double attribute representing the balance.



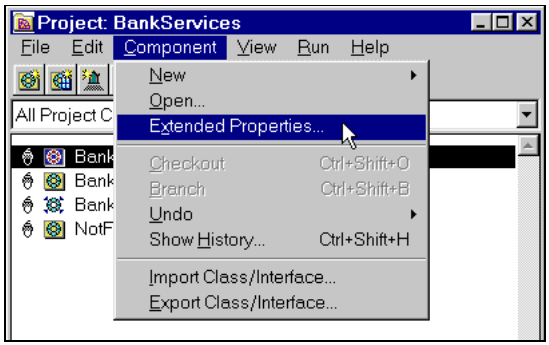
If you want to export a method that uses the BankAccount object as a parameter or return value, then you need to mark the object as an XMLStruct, as shown in the following procedure. The methods of an object being exported as an XMLStruct are ignored during the conversion to a compound data type. However, the Init method is still called by the XML server when it instantiates the object.

➤ **To mark an object as an XMLStruct**

1. In the Repository Workshop, open your project and select the object you want to mark as an XMLStruct.

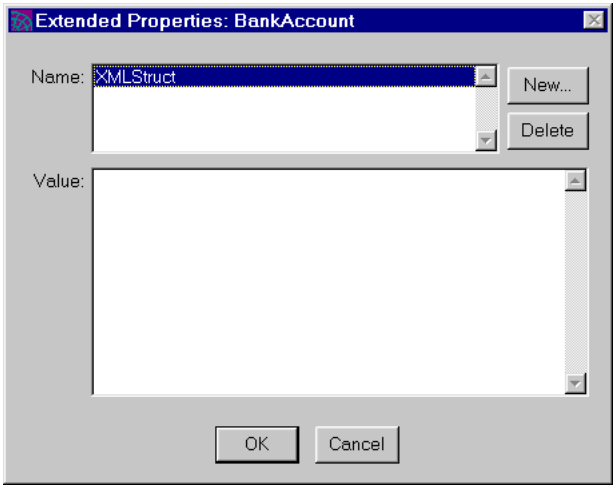


2. In the Project Workshop, select Component > Extended Properties.

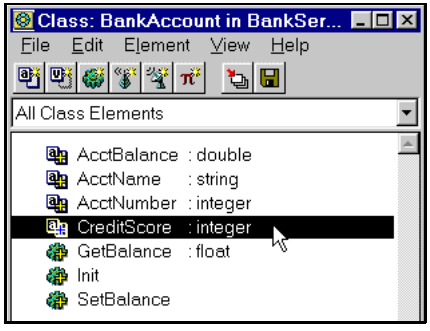


3. In the Extended Properties window, select New, specify “XMLStruct,” and click OK.

   Leave the Value field blank. You are not required to set the value for the XMLStruct property.



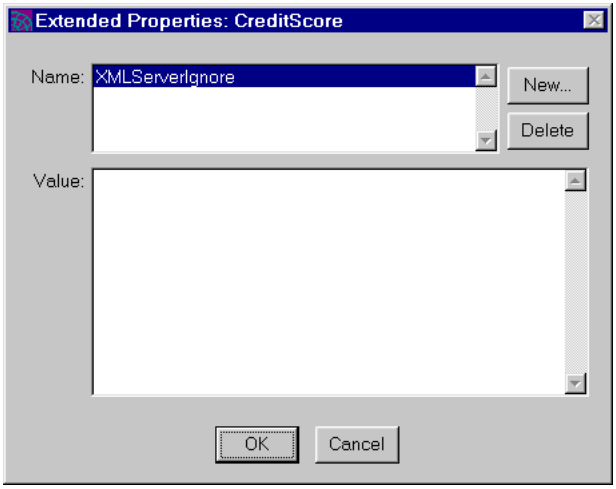
4. If the object you are exporting has a public attribute you do not want to export, then in the Class Workshop, select the attribute.



5. Still in the Class Workshop, select Element > Extended Properties.

6. In the Extended Properties window that opens, select New, specify "XMLServerIgnore," and click OK.

   Leave the Value field blank. You are not required to set the value for the XMLServerIgnore property.



Procedures for Exporting an XML Server

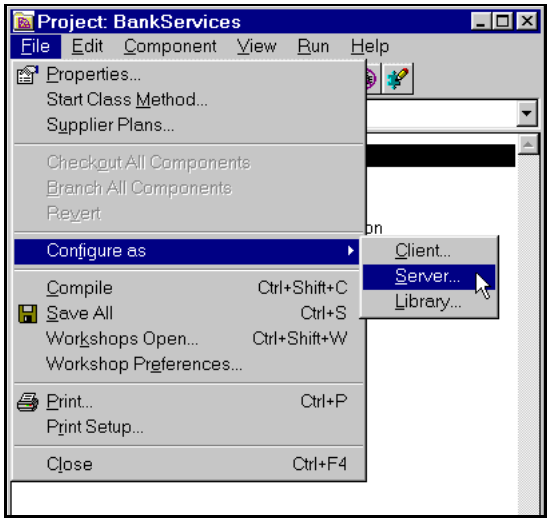
After you have defined the properties of a service object, you are ready to export it as an XML server using either the Repository Workshop or Fscript commands.

Exporting Using the Repository Workshop

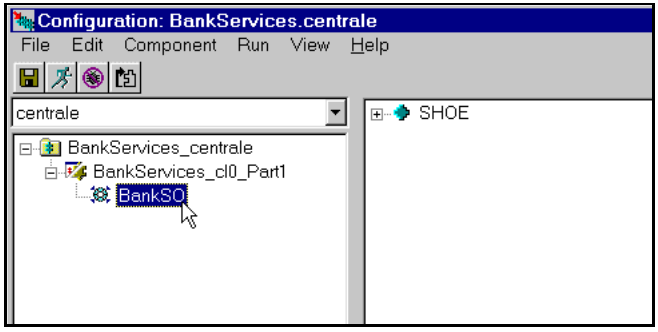
The following procedure shows how to export an XML server using the Repository Workshop. The procedure illustrates exporting the BankServices project, an example provided in the iPlanet UDS examples directory.

➤ **To export a service object as an XML server**

1. In the Project Workshop, select File > Configure As > Server.

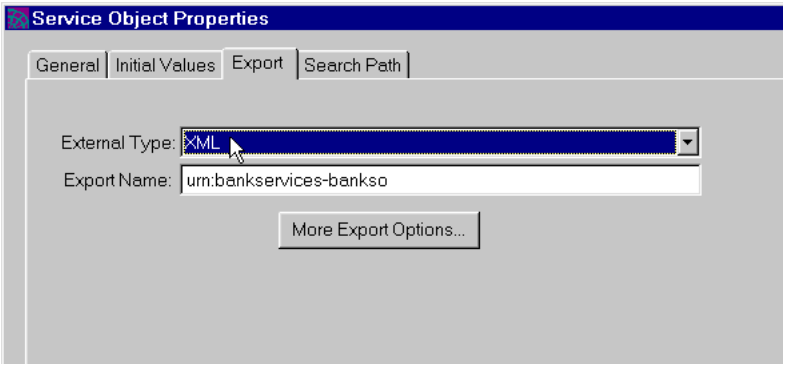


2. In the Configuration Window that opens, expand the project to select the service object.



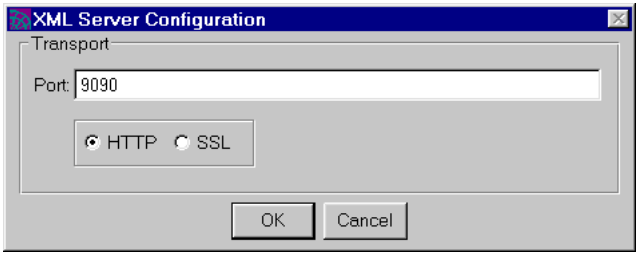
3. Still in the Configuration Window, select Component > Properties.

4. In the Service Object Properties window that opens, click the Export Tab and specify the External Type as XML.



5. In the More Export Options Window that opens, specify the protocol settings.

   The default port is 9090 and the default protocol is HTTP. You can override these default values. If you specify SSL, the default port is 9443.



6. Still in the Service Object Properties Window, you can override the default Export Name.

   The Export Name identifies the server for access by client applications. By default, the Export Name is built from the project name and the service object name, and has the following form:

   `urn:`*ProjectName-ServiceObjectName*

   The "urn:" prefix helps define a unique namespace for the server.

7. In the Configuration Window, select File > Make Distribution

   Specify the appropriate settings for your environment to make a distribution of your XML server application. When you make the distribution, a WSDL file describing the service object is generated at the following location:

   `FORTE_ROOT/appdist/`*`envname`*`/`*`distribID`*`/cln/codegen/`

NOTE Generating Java files for client applications is a separate procedure, described in "Generating Files for Java Clients" on page 152.

Exporting Using Fscript

You can export a service object as an XML server using Fscript commands. Code Example 7-1 illustrates the commands you can use to export the BankServices example provided with your iPlanet UDS distribution. This example does not override any of the default values for XML services. The following section, "Overriding Default Values," shows how to override the default values.

Code Example 7-1 Exporting a service object as an XML server using default values

```
FindPlan BankServices
-- Set the Project as a server without a client partition
SetProjType 2
Partition

-- Specify to export as an XML Server
SetServiceEOSInfo BankServices.BankSO xml
Partition

-- Create a new application distribution on node garf
-- (without auto-recompile), and install the application
MakeAppDistrib 1 garf 0 1

Partition
Commit
Close
```

NOTE Generating Java files for client applications is a separate procedure, described in "Generating Files for Java Clients" on page 152.

Overriding Default Values

Code Example 7-1 on page 147 exports the XML server application using the following default values for the server attributes:

Export Attribute	Default Value
Export Name	urn:*ProjectName-ServiceObjectName*
Transport	http
Host	localhost
Port	9090
Protocol	SOAP

Use SetServiceEOSInfo to override the export name. The syntax for SetServiceEOSInfo to specify an XML server is:

```
SetServiceEOSInfo service_object_name XML [export_name]
```

export_name specifies the export name.

Use SetServiceEOSAttr to specify the server attributes for the protocol, transport, and port number. You make multiple calls to SetServiceEOSAttr to specify each attribute. The syntax for SetServiceEOSAttr is:

```
SetServiceEOSAttr service_object_name key value
```

key is the attribute you want to set. *value* is any of the allowed values for the attribute specified by *key*. Table 7-5 lists the possible key/value pairs for SetServiceEOSAttr.

Table 7-5 Key/Value pairs for SetServiceEOSAttr

Attribute (Key)	Possible Values	Comments
ProtocolType	SOAP	Currently, only protocol supported.
TransportHost	*hostname*	Fully-qualified domain name.
TransportPort	*PortNumber*	
TransportType	http ssl	

Code Example 7-2 illustrates an example that overrides some of the default XML server attributes.

Code Example 7-2 Overriding default XML server attributes

```
. . .
-- Export as an XML Server
SetServiceEOSInfo BankServices.BankSO xml urn:my-xmlserver

SetServiceEOSAttr BankServices.BankSO TransportType ssl
SetServiceEOSAttr BankServices.BankSO TransportHost garf.forte.com
SetServiceEOSAttr BankServices.BankSO TransportPort 8888
. . .
```

Starting an XML Server

After exporting an XML server, you can either start the server manually or use iPlanet UDS system management tools to start the server. This section contains examples of using iPlanet UDS system management tools to start and stop an XML server.

For information on starting server partitions manually, refer to the chapter on managing iPlanet UDS Applications in the *iPlanet UDS System Management Guide*.

NOTE You cannot rely on the iPlanet UDS auto-startup technique to start the server partition. Auto-startup specifies that the server partition is started only when required by a client partition. iPlanet UDS XML servers must be supported manually to support access by external applications.

Starting and Stopping an XML Server

Code Example 7-3 shows an example of Escript commands that illustrates how to start the BankServices example supplied with your iPlanet UDS distribution. You can run this example after you export the BankServices project as an XML server.

For information on starting and managing iPlanet UDS server applications, refer to the *Escript and System Agent Reference Guide* and the *iPlanet UDS System Management Guide*.

Code Example 7-3 Using Escript to start an XML server

```
findsub BankServices_cl0
start
exit
```

Code Example 7-4 shows how to shutdown the server that was started in the previous example.

Code Example 7-4 Using Escript to stop an XML server

```
findsub BankServices_cl0
shutdown
exit
```

Chapter 8

Creating Java Client Applications for an XML Server

This chapter shows how to create Java client applications that access an iPlanet UDS XML server.

About Java Client Applications

After exporting a service object as an XML server, you can generate Java source files that enable Java client applications to access the server using the Simple Object Access Protocol (SOAP). You can also generate Java files that allow you to pass SOAP complex datatypes to and from the XML server. Additionally, when you make an application distribution for an XML server, WSDL files that describe the server are automatically generated.

Generating Files for Java Clients

Use the Fscript command GenerateXMLServerJava to generate the Java source files. Generating Java source files can only be done using Fscript commands.

Code Example 8-1 shows how to generate the Java files for the BankServices example, included with your iPlanet UDS distribution.

Code Example 8-1 Generating Java source files

```
FindActEnv
findplan BankServices
GenerateXMLServerJava
Commit
```

NOTE You should only call GenerateXMLServerJava after you have made an application distribution of an XML server. For information, refer to Chapter 7, "Exporting an iPlanet UDS Service Object as an XML Server."

The GenerateXMLServerJava command generates the following Java source files:

*ServiceObject*Proxy.java (one file for each service object exported)
CompoundDatatype.java (one file for each SOAP compound data type)

Service Object Proxies

*ServiceObject*Proxy.java is an XML server proxy that provides Java methods corresponding to exported Java methods in the UDS service object. To access an exported service object method, simply call the corresponding method in the proxy. For each exported service object, UDS generates a Java proxy file. By default, proxies connect to the service object at the following location:

```
http://localhost:9091
```

A proxy contains a constructor that allows access to specified hosts.

JavaBean Files for Compound Data Types

A *CompoundDatatype*.java is generated for each object in the XML server that is defined as a SOAP complex data type (marked as an XMLStruct). This file defines a JavaBean for the object and thus contains accessors and mutators for each member of the compound data type. Use an instance of this JavaBean as a parameter or return value to proxy methods that access the corresponding object in the XML server.

Location of Generated Java Files

GenerateXMLServerJava places the generated files in a Java project at the following location:

```
FORTE_ROOT/appdist/envname/projname/generic/java
```

envname represents the name of your environment. *projname* is a generated name based on the names of the project.

The Java project has the following directory structure, with the generated Java files placed in the *projectname* directory:

```
com/forte/xmlsvr/examples/projectname
```

Generated WSDL Files

When you export an iPlanet UDS service object as an XML server, a WSDL file describing the service object is automatically created and placed in the following location:

```
FORTE_ROOT/appdist/envname/distID/cln/codegen/proj
```

envname represents the name of your environment. *distID* and *proj* are based on the distribution ID and the name of your XML server project.

Using the Generated Proxy

The generated proxy file can be used with client applications without any further editing. However, you can edit the generated file as appropriate for your specific needs.

The proxy files depend on a CLASSPATH that includes the following locations:

- SOAP library
- XML parser library
- Java Software Development Kit (JDK)

For information on supported versions of SOAP and related software, refer to the Platform Matrix available at http://www.forte.com/support/platforms.html.

Generated proxies contain constructors and additional methods corresponding to methods exported by the service objects.

Proxy Constructors

The constructors for a proxy takes the following forms:

ServiceObjectName`Proxy()`
ServiceObjectName`Proxy(String url)`

ServiceObjectName is the name of the service object. *url* is the location of the XML server. If you do not specify *url*, then the proxy attempts to connect to the XML server using the following default URL:

```
http://localhost:9091/
```

Proxy Methods

Each method in the proxy corresponds to methods in the exported service object. For example, if the service object contains the method:

```
CreateAccount(BankAccount acct): integer
```

Then the corresponding proxy method is:

```
public int CreateAccount(BankAccount parm_acct) throws Exception
```

Each proxy method automatically handles the details of making SOAP requests to the XML server. A proxy method does the following:

- Serializes objects passed to the server using a fully-qualified name generated by the server.
- Builds an argument vector for all parameters passed to the server.
- Builds a call to the server using the generated name, serialized objects, and argument vector.
- Sends the call to the server using the URL defined in the constructor.
- Throws a Java exception if the SOAP call generated a SOAP fault response.

  The Java client must catch the exception, which contains information about the SOAP fault.
- Captures any return value in an object (which is cast to the appropriate Java type for the return value)
- Returns the result (if the method returns a value)

Using the Generated JavaBean

Your XML server may include exported objects that represent SOAP compound data types (marked as XMLStructs in the UDS service object). For each XMLStruct, GenerateXMLServerJava generates a Java source file that creates a JavaBean to define the XMLStruct.

The Java source file contains a constructor for the exported object, as well as accessor and mutator methods for each element in the object. You should not modify this file—it provides your client access to complex objects in the server.

Use the JavaBean to instantiate SOAP compound objects that are included in SOAP messages, either as method parameters or return values.

Using the Generated WSDL File

The generated WSDL file is an XML document that describes the XML service object according to the WSDL specification. You typically use this file as input to a third party utility that can generate SOAP client applications, or to publish the exported services on an external registry.

XML Server Examples

The BankServices example provided with your iPlanet UDS distribution illustrates an XML server that provides a basic implementation of a banking service. It is based on the sample application you build with the iPlanet UDS tutorial in the *Getting Started With iPlanet UDS* manual.

The BankServices example allows you to display account information, make deposits and withdrawals, and add new accounts. The example provides two sample Java client applications that illustrate how to use the generated proxy and JavaBean files.

You can find the BankServices example at the following location in your distribution:

```
FORTE_ROOT/install/examples/extsys/xmlsvr/
```

Part 3

Using External C Functions

Part 3 of *Integrating with External Systems* provides complete information about the C data types you can define in TOOL when you are integrating with external systems. It also provides complete information about integrating with C.

Part 3 contains the following chapters:

Chapter 9

Encapsulating External C Functions

Many development organizations use existing C libraries as an important part of their application development processes. In fact, your organization might require standardized data types and operations as a part of every application. You also might find that your application can perform specialized processing more efficiently in languages or systems external to iPlanet UDS.

iPlanet UDS lets you create classes whose methods are implemented with C functions. These classes are stored in the repository as C projects, and you can use these methods in your TOOL applications just like any other method.

This chapter discusses the following topics:

- rules and restrictions for C projects
- creating C projects and linking them to the external functions
- installing C projects
- writing TOOL methods that invoke C functions
- reference information about the TOOL statements for creating C projects, classes, and methods

About Encapsulating External C Functions

This discussion of integrating iPlanet UDS applications with external C functions assumes that you have a set of compiled *object modules* that you want to integrate. Depending on your environment, you may refer to these modules as object libraries or archives, shared images, shared libraries, or DLLs.

This chapter uses the term *C functions* to refer to any external routines that can be invoked using C calls.

This section provides an overview of creating projects, classes, and methods for integrating external C object modules with iPlanet UDS applications. Also discussed is the concept of restricted projects and their impact on installation.

Terminology Used in Part 3, "Using External C Functions"

The following list defines terms used in this chapter that might have synonyms on different platforms:

iPlanet UDS linked executable Result of statically linking iPlanet UDS with the user object modules.

iPlanet UDS object module File for the C++ wrapper code that you compiled. iPlanet UDS generates this C++ wrapper code when you make a distribution.

Shared library Result of dynamically linking iPlanet UDS object modules with user object modules. On some systems, this is referred to as a shared library (UNIX), or a DLL (PC).

User object module Compiled C functions. On some systems, these are referred to as object libraries, object archives, shared images, shared libraries, or DLLs.

Accessing C Functions from Within iPlanet UDS Applications

This section provides an overview of creating projects, classes, and methods for using C functions in iPlanet UDS applications. For detailed instructions, see "About Making C Functions Available to iPlanet UDS Applications" on page 165.

Using iPlanet UDS, you can write applications that access any C functions available in your installation. To make these C functions available to an iPlanet UDS application, you create an iPlanet UDS project, referred to as a *C project*.

A C project contains classes whose methods are named the same as the C functions. In C project classes, each method directly corresponds to a C function. When you invoke the method, iPlanet UDS calls the associated C function.

When you create and change C projects, you must work in a text file, the *project definition file*, then import this file into the repository to create or change the C project in the repository. You cannot use the Project Workshop to create or modify C projects.

After you import the C project, you partition the project, then make a distribution to generate C++ *wrapper code*. This wrapper code implements the iPlanet UDS methods by calling the associated C function. The wrapper code must be compiled with a C++ compiler and then linked with the C shared library. For every platform, the compiler you use to compile the C++ code must be the same compiler used by iPlanet UDS. The compiling and linking steps can be automated if your iPlanet UDS system manager sets up your environment appropriately.

Finally, within Environment Console or Escript, you or your iPlanet UDS system manager loads and installs the distribution onto the appropriate nodes. Again, this step can be automated if the iPlanet UDS system manager sets up your environment appropriately.

You can use a C project class in TOOL code like any other class. You can create objects of a C class type, and you can use the parameters, belonging to that class type. To call a C function, you create an object of the class type and then invoke on that object the method associated with the C function.

NOTE A C project is restricted if it can run on only a subset of nodes in your environment. If a TOOL project uses a class that creates an instance of a restricted C class, then that TOOL class must also be restricted. See the *TOOL Reference Guide* for more information about the `restricted` project property.

TOOL Statements for Defining C projects

iPlanet UDS provides special versions of some TOOL statements to let you define C projects: **begin c** and **class**. The complete syntax for these statements is included in "begin c" on page 192 and "class" on page 198.

Prepare to Wrap C Functions

This section describes some considerations and setup tasks to perform before you wrap C functions.

Set up the Auto-Compile Application

If your environment is set up for auto-compiling, you can compile, link, and install the shared libraries on the appropriate nodes when you make the distribution.

You can only use the auto-compile application on server partitions or platforms running Windows 95. For information about setting up the environment for auto-compiling, see the *iPlanet UDS System Management Guide*.

Can or Should the C Project Be Multithreaded?

By default, the multiThreaded property of the C project is TRUE, which means iPlanet UDS will attempt to run multiple threads through the wrapped object module. On Windows NT, you must leave the multiThreaded property as TRUE, which means that you must always be sure to link with a version of the standard C library that is thread-safe.

You should considering the following issues to determine whether you can safely run the C project multithreaded:

- If you are deploying the C project on UNIX or VMS, the wrapped object module must be interrupt safe.

  In iPlanet UDS, thread-switching is scheduled using a periodic interrupt such as a UNIX signal or VMS AST. These asynchronous notifications might interrupt certain system calls started by a TOOL application that is using the wrapped object module. The wrapped object module should wrap each system call in a loop that will retry the call if that call is interrupted. Here is an example:

```
do
{
  sysret = recv(...);
} while (sysret < 0 && errno == EINTR);
```

  Because iPlanet UDS uses native NT threads on Windows NT, interrupts of system calls are handled by the Windows NT threading system.

- The wrapped object module must be re-entrant, in that the wrapped object module should not let different threads change the same data structures at the same time.

  If the wrapped object module does not prevent this unintended sharing, you need to set the multiThreaded property to FALSE, or use a Framework Mutex object to prevent the TOOL code from starting multiple threads in the wrapped object module.

  If you are using this wrapped object module on a Windows NT node, then you must make the wrapped object module re-entrant to use it on Windows NT.

- If the standard C library is not thread-safe, then the wrapped object module must not use the standard C library functions.

  If the wrapped object module makes input and output or other standard C library calls that allocate memory or utilize shared data, you must set the multiThreaded property for the C project “multiThreaded = FALSE”. Many standard C libraries are not thread safe, so it is not safe for the wrapped object module to enter them when another thread inside the iPlanet UDS runtime might already be using the same function or memory in the standard C library.

- If you are deploying the C project on UNIX or VMS, the wrapped object module should avoid making system calls that block, especially when the project is marked "multiThreaded = FALSE".

  A system call that blocks when it performs input or output or accesses other system resources will block the entire server partition if the system call needs to wait to complete. If the blocking system calls are made interrupt safe, as described above, and the project is marked "multiThreaded=TRUE," you can use blocking system calls. In this case, a TOOL invocation of a wrapper method blocks because of an interrupted system call until the iPlanet UDS runtime system switches threads. When the iPlanet UDS runtime system switches back to this thread, the loop surrounding the system call should retry the system call, as described above.

  One way to prevent a C project with "multiThreaded=FALSE" set from blocking the entire application is to define a service object with methods that invoke methods on this C project. By placing this service object in its own partition, you can cause only this partition to block when necessary, allowing the partition that provides most of the application's services to proceed normally.

Turning off multithreading for a C project:

- Disables thread-switching by the iPlanet UDS runtime system for tasks that use methods defined in the C project.
- Automatically locks a mutex upon entry into the C project and unlocks the mutex on exit.
- Coordinates single-threaded access to the standard C library with the rest of the iPlanet UDS runtime system.

Make Sure the Proper C++ Compiler Is Installed

For information about the C++ compiler that you should have installed for each platform, see the *iPlanet UDS System Installation Guide*.

Chapter 10

Making C Functions Available to iPlanet UDS Applications

This chapter explains how you define a C project whose methods map to your C functions.

Topics covered include:

- making C functions available to iPlanet UDS applications
- making the distribution with auto-compile and auto-install
- updating C projects
- making installed C projects known to other repositories

About Making C Functions Available to iPlanet UDS Applications

After you have defined and imported this C project, you can write TOOL applications that use the C functions.

This chapter assumes that the C functions are installed, tested, running, and available to iPlanet UDS.

➤ **To integrate an iPlanet UDS application with external C functions**

1. Make the C object modules available.
2. Create a C project using a subset of TOOL project definition statements.
3. Import the C project definition into your repository.
4. Partition the C project.

5. Make the distribution to generate the C++ wrapper.
6. Compile and link the shared libraries.
7. Install in appropriate platforms.

If your environment is set up for auto-compiling and installation, you can combine Step 5, Step 6, and Step 7 to compile, link, and install the shared libraries on the appropriate nodes when you make the distribution (see "Making the Distribution with Auto-Compile and Auto-Install" on page 174).

Each step is described in detail in this section.

Static Loading Platforms

Certain versions of the Sequent operating system do not support dynamic linking. On this platform you must execute commands to link complete iPlanet UDS executables containing your C projects. After you make the distribution and compile the C++ wrapper code, you will need to link one or more iPlanet UDS executables instead of linking a single shared library. For example, if you want to run your C project from Fscript, then you must link an Fscript script that includes your C project. You cannot automatically link, compile, and install C project shared libraries on these platforms.

Call your iPlanet UDS Technical Support representative for instructions.

Examples

The examples in this section are based on the sample program DMathTm. DMathTm shows how you can write a C project definition that makes some ANSI C standard library functions available to TOOL programs.

Have the Object Modules for the C Functions

Any C functions you want to integrate with iPlanet UDS applications need to meet certain requirements:

- Some platforms require C object modules to be shared libraries, which are position-independent. Check the following table to determine whether position-independent code is required for your platform.

Platform	Rules for compiling C functions
Data General	The C functions that you integrate must be compiled with the following flags: cc -c -K PIC *.c
Digital UNIX	The C functions that you integrate must be compiled position-independent. By default, the DEC Digital Unix C compiler compiles position-independent code.
HP 9000	The C functions that you integrate must be compiled position-independent. The flag to "cc" that produces position-independent code is the "+z" or "+Z" flag. The standard C compiler does not support this flag. You must get the optional C compiler that supports the generation of position-independent code.
RS6000	The C functions that you integrate should be compiled with the -qchars=signed flag if you want TOOL il values to be signed.
Sparc	The C functions that you integrate must be compiled position-independent. The flag to "cc" that produces position-independent code is the "-PIC" flag.
VMS	By default, the DEC C compiler compiles position-independent code.
Windows	The C functions must be compiled for use within a DLL using the large memory model.

- iPlanet UDS assumes that the C functions are defined with prototypes, as in ANSI C.

  If the C functions were not defined using prototypes, you cannot use the auto-compile feature when making a distribution, because an additional flag is required at compile time. For more details see "Compile and Link Shared Libraries" on page 177.

- If you plan to use the auto-compile feature when making a distribution, you need to specify a directory in your C project definition **extended** external properties and copy the C object modules to this directory. iPlanet UDS can then automatically include this directory as a linking option when it compiles and links the iPlanet UDS object modules into shared libraries.

  For more information about the **extended** external properties, see "Defining Properties" on page 171 and "Extended External Properties" on page 195.

Create the C Project Definition File

The project definition file is a text file in which you define the C project that will integrate your C functions. Like all TOOL projects, the definitions for a C project can be contained in a single file or within several files. Within a C project you can specify constants, classes, service objects, and the definitions of derived data types. Within a class, you can create constants and methods, and set the properties of the class.

C Project Class Restrictions

The following restrictions apply to C project classes:

- You cannot subclass C classes.
- C classes cannot contain attributes or events.
- The data types of class method parameters and method return values cannot be object classes.

Defining a Project

begin C statement

You must create a C project using a `begin C` statement in a TOOL file. You cannot use the Project Workshop to create or edit C projects, because C projects are imported into the repository as read-only projects. To define the project components, you use the `class` statement and `constant` statement. The C project `class` statement is a subset of the `class` statement defined in the *TOOL Reference Guide*. The syntax for the `begin c` and `class` statements specific to C project classes can be found in the following sections:

- "begin c" on page 192
- "class" on page 198

There are no rules about the ratio of C functions to classes. Each function can have its own class, or all functions can be placed in one class. Generally, you'll probably want to collect a set of common operations and place them in one class. A project can have any number of classes.

Service Objects

You can use the `service` statement to declare a service object within your C project definition using a class in the C project. Defining a service object using a C class means that the C functions mapped to that class are available to multiple users. This type of service object is particularly useful if your C project is not restricted and the C functions associated with the C class use only simple data types as their parameters.

For more information about the `service` statement, see the *TOOL Reference Guide*.

Supplier C projects

You can use the `includes` clause to specify a C project that is a supplier project for this C project. You can specify one or more `includes` statements. Only C projects can be supplier projects for a C project.

For more information about the includes clause in a C project definition, see "begin c" on page 192.

Derived data types You can also define derived C data type definitions in your project, including enums, structs, typedefs, and unions. For more information about these data types, see "Using C Data Types in TOOL Methods" on page 199.

Example: C Project File

```
---------- Dmathtm.pex ---------------------
begin C DistMathAndTimeProject;
has property restricted = TRUE;
-- This TOOL struct matches the struct tm in the standard header
-- file time.h.
 struct tmstruct
  tm_sec : int;
  tm_min : int;
  tm_hour : int;
  tm_mday : int;
  tm_mon : int;
  tm_year : int;
  tm_wday : int;
  tm_yday : int;
  tm_isdist : int;
 end;
typedef time_t : int;
class timeFunctions
  OurTime() : time_t;
  OurLocalTime(input a : pointer to time_t) : pointer to tmstruct;
  OurAscTime(input a : pointer to tmstruct) : string;
end;
class mathFunctions
  OurExponent(input a : double) : double;
  OurPower(input x : double, y : double) : double;
end;
has property
  LibraryName = 'dmathtm';
  Extended = (ExternalSharedLibs = '/usr/shlib/libc',
      ExternalObjectFiles = '%{FORTE_ROOT}/tmp/examples/dmathtm');
UUID = 'EAEE2938-F769-11CE-B998-7842A7C4AA77';
end DistMathAndTimeProject;
```

File: dmathtm.pex • **Project:** DistMathAndTimeProject

In this example, the project is called DistMathAndTimeProject, and it contains two classes—`timeFunctions` and `mathFunctions`. The `timeFunctions` class has the methods `time`, `localtime`, and `asctime`. The `mathFunctions` class has the methods `exp` and `pow`.

In the above example, the `libraryname` value for the project is "`dmathtm`." If specified, the `libraryname` property defines the name of the iPlanet UDS object module for this project. The value for `libraryname` must be different than the file names used for the user object modules.

The above example contains two extended properties:

```
Extended = (ExternalSharedLibs = '/usr/shlib/libc',
      ExternalObjectFiles = '%{FORTE_ROOT}/tmp/examples/dmathtm');
```

The `ExternalSharedLibs` property specifies a library and the `ExternalObjectFiles` property specifies an object file that will be included as linking flags when the shared library for this project is linked.

For more information about the extended properties, see "Extended External Properties" on page 195. For more information about linking flags, see "Compile and Link Shared Libraries" on page 177.

Defining Properties

In C projects, you can specify the following properties:

Property	Description
restricted	TRUE specifies that the C project can be partitioned only on certain nodes in the development environment. FALSE means that the C project can be partitioned on all nodes.
multithreaded	Specifies whether your project is thread-safe and signal-tolerant.
compatibilitylevel	Specifies the compatibility level for the project.
extended external properties	Specify files to be included as linking options or header files when the compiled shared libraries are linked. If you plan to use the autocompile feature when you make the distribution, you need to specify the directories that will contain the C object modules.
libraryname	Specifies the name of the iPlanet UDS object module and the component ID.

For more information about these properties, see "Has Property Clause" on page 193.

Defining a Method

The following are the rules for mapping a C function to a TOOL method:

- Name the mapping method exactly the same as the C function—be sure to match the case.

  Each method definition you create for your C project must map directly to a C function. The name of the function must match exactly the name of the method; method naming is case-sensitive.

  If you have two C functions with the same name, differing only in the case used in the name, for example, myfunction and MYFunction, TOOL treats them as the same name. In this case, you must write a wrapper C function that calls one of the functions, and use the name of this wrapper function in the C project.

- C mapping method names must be unique within the application. If you have ProjectA and ProjectB, both of which are C projects, you must not use the same method name in both C projects. All methods must have unique names. (On some platforms, duplicate names produce a link error when compiling; on other platforms, one of the methods is selected at runtime based on the linking order.)

- Name the TOOL method parameters and define their data types based on the C parameter types (see Chapter 13, "Using C Data Types in TOOL").

- Specify the return type.

Let's examine the localtime C function to illustrate how it maps to a TOOL method. The localtime C function has the following function header:

```
struct tm *localtime(const time_t *tp)
```

The corresponding TOOL method is defined as follows:

```
localtime(input a : pointer to time_t) : pointer to tmstruct;
```

The mapping is fairly straightforward:

- The method is named "localtime", exactly like the C function.
- The C function parameter "const time_t *tp" maps to "a : pointer to time_t" in TOOL.
- The TOOL method parameter is an `input` parameter, because the C function returns its output using the return value, not with the parameter.
- The return type "struct tm *" maps as "pointer to tmstruct", where tmstruct is the TOOL data structure that maps to tm.

For more information about mapping C function parameters to TOOL method parameters, see Chapter 13, "Using C Data Types in TOOL."

Import the C Project Definition File

Once you complete the project definition, you import the file using the Fscript `ImportPlan` command or the Import command in the Repository Workshop.

Importing the C project creates a project in your iPlanet UDS repository that contains the classes and methods you defined in the C project definition.

For more information about using the Repository Workshop, see *A Guide to the iPlanet UDS Workshops*. For more information about Fscript, see the *Fscript Reference Guide*.

Partition the C Project

The iPlanet UDS system knows that the C project is a shared library, and represents it with a library icon in the Repository Workshop. If this C project is a restricted project, you should assign this library only to nodes where the C functions are installed and running.

➤ **To partition your C project as a library**

1. Open the Project Workshop for the C project.
2. In the Project Workshop, choose the File > Configure As Library command.

   iPlanet UDS displays the default partitioning for the library in the Partition Workshop.
3. You should remove the library from nodes that do not have the C functions installed and running.

You can also use the Fscript command `Partition`. You can remove the project from nodes where you do not want this library installed.

For more information about partitioning libraries, see *A Guide to the iPlanet UDS Workshops*.

Make the Distribution

Next, you need to make a distribution to generate the files you need for the configuration you specified. You can make the distribution by issuing the `MakeAppDistrib` command in Fscript or by making the distribution from the Partition Workshop. For more information about Fscript, see the *Fscript Reference Guide*. For more information about the Partition Workshop, see *A Guide to the iPlanet UDS Workshops*.

If your environment is set up for auto-compiling, you can compile, link, and install the shared libraries on the appropriate nodes when you make the distribution.

If you choose not to use the auto-compile and auto-install features, you can find the steps for compiling, linking, and installing the shared libraries without using the automated features, in the following sections:

- "Compile and Link Shared Libraries" on page 177
- "Install C Project Shared Libraries" on page 181

Making the Distribution with Auto-Compile and Auto-Install

Your iPlanet UDS system manager can set up your system so that you can automatically compile and link the C project into shared libraries ("Compile and Link Shared Libraries" on page 177) as part of "Make the Distribution."

NOTE If your C functions are defined without prototypes, you cannot auto-compile and install when making a distribution, because you need to specify special flags on the `fcompile` command. For more information about compiling and linking for C functions that do not have prototypes, see "C function without prototype" on page 180.

The steps for setting up the system for auto-compile and auto-install are explained in the *iPlanet UDS System Management Guide*. In general, your system manager must set up the following components on your system:

- one or more code generation servers to generate the code for the distribution
- a server that manages how and where shared libraries are compiled and linked

- one auto-compilation server for each platform where the shared libraries for the C projects will be installed. Each of these servers must have access to the C++ compiler for that platform.

If your system manager has set ups these components, then you can make the distribution with the auto-compile feature to perform the following steps automatically:

- Create a distribution directory structure for the current configuration of the current C project.
- Generate C++ wrapper code.
- Compile and link the C++ wrapper code into the shared library required for each platform.
- Place the shared libraries into the appropriate distribution directories, as described in "Make the Distribution" on page 174.

If you also selected auto-install, making the distribution also installs the shared libraries on the appropriate nodes in the development environment, according to the configuration you specified when you partitioned the C project.

➤ **To make a distribution with auto-compile and auto-install**

1. After you have partitioned your C project as a library, choose the File > Make Distribution command.
2. In the Make Distribution dialog, select Partial Make (to update a distribution) or Full Make (to create a new distribution), then select the toggles for Install In Current Environment and Auto Compile.

3. Select the Make button.

You can also use the `MakeAppDistrib` command in Fscript to make a distribution with auto-compile and auto-install, as shown:

```
fscript> MakeAppDistrib 1 "" 1 1
```

For more information about making a distribution, see *A Guide to the iPlanet UDS Workshops*, and for information about the Fscript `MakeAppDistrib` command, see *Fscript Reference Guide*.

You can now skip the rest of this section and start writing TOOL clients, as described in Chapter 11, "Writing TOOL Code That Uses C Functions."

Making the Distribution without Auto-Compiling

If you are making the distribution without using the auto-compile feature, then making the distribution simply generates the code you need to produce shared libraries for the current configuration of this C project. You need to use additional iPlanet UDS utilities to compile and link the generated C++ wrapper code files to produce the shared libraries. Then, you need to place these shared libraries in the appropriate distribution directories to enable the iPlanet UDS system manager to automatically install these shared libraries in the right places in your environment.

When you make a distribution without auto-compile and auto-install, the distribution places files in the following directories:

`FORTE_ROOT/appdist`*/environment_id/distribution_id/cl#/codegen/component_id*

`FORTE_ROOT/appdist`*/environment_id/distribution_id/cl#/generic/component_id*

Directory name	Description
environment_id	First 8 characters of the environment name where you want your C project to be installed.
distribution_id	8 character name derived from the C project name.
cl#	Compatibility level for this project, as specified as the value of property `compatibilitylevel` in the C project definition.
component_id	Value of the property **libraryname** in the C project definition, or the first 8 characters of the project name if `libraryname` is not specified.

The iPlanet UDS system places the following files in the `codegen/`*component_id* directory:

- Two C++ wrapper code files that must be compiled and then linked with the C functions to produce the shared library. The names of these files are based on the `libraryname` value for the project and have the extensions `.cc` and `.cdf`. For example, if the library name is dmathtm, then the files are "dmathtm.cc" and "dmathtm.cdf." If `libraryname` is not specified, the file names are the first 8 characters of the project name.
- A file listing all the files needed to compile this component, which has an extension of .bom.
- If you make the distribution on VMS or MS-Window platforms, you must be aware of two files in addition to the.cc and.cdf files:

Platform	File extensions	Examples
VMS	.mar	"dmathtm.mar"
	.opt	"dmathtm.opt"
Windows	.def	"dmathtm.def"
	.exp	"dmathtm.exp"

In the `generic/component_id` directory, the iPlanet UDS system places a .pex file that you can use to import the C library into a repository that does not contain the library source code. When you install this library, this .pex file is copied to the directory where the library distribution is installed:

`FORTE_ROOT/userapp/`*distribution_id*`/cl#`.

Compile and Link Shared Libraries

After you make a distribution for the C project, you must create a shared library for each platform where the C project shared library will be installed. This shared library is later dynamically loaded into the iPlanet UDS system where it is used to access the C functions.

You need to compile and link the shared libraries for each platform where this project's shared library is installed.

➤ **To compile and link the shared libraries**

1. Copy the generated distribution files to a node with the same platform as one or more of the nodes where the C project shared library will be installed. This node must have the required C++ compiler so that the code can be compiled and linked.

   The files you need to copy are in the directory path described under "Make the Distribution" on page 174.

2. Use `fcompile` to compile and link the generated code and libraries into a shared library.

   The syntax of the `fcompile` command when you compile a C project library is:

Portable syntax (all platforms)

fcompile [**-c** *component_generation_file*] [**-d** *target_directory*]
[**-cflags** *compiler_flags*] [**-lflags** *linking_flags*]
[**-fm =** *memory_flags*] [**-fl =** *logger_flags*] [**-cleanup**]

OpenVMS syntax

VFORTE FCOMPILE
[**/COMPONENT =** *component_generation_file*]
[**/DIRECTORY =** *target_directory*]
[**/COMPILER =** *compiler_flags*]
[**/LINKING =** *linking_flags*]
[**/MEMORY =** *memory_flags*]
[**/LOGGER =** *logger_flags*]
[**/CLEANUP**]

The following table describes the command line flags for the `fcompile` command:

Flag	Description
-c *component_generation_file* **/COMPONENT =** *component_generation_file*	Specifies the file that iPlanet UDS compiles. This value includes the path where the file resides if the file is not in the current directory. By default, iPlanet UDS compiles all files in the current directory. The *`component_generation_file`* value for a C project has a file name that is either the value of the property `libraryname` in the C project definition or the first 8 characters of the project name, if `libraryname` is not specified. This file has the extension.bom.

Flag	Description
-d *target_directory* **/DIRECTORY =** *target_directory*	Specifies where the compiled directories will be placed. By default, `fcompile` compiles files in the current directory, and places the compiled files in the current directory. *target_directory* is a directory specification in local syntax. If the **-c (/COMPONENT)** flag is also specified, the **-d** flag specifies where the compiled component files will be placed. Otherwise, the directory specified by the **-d (/DIRECTORY)** flag specifies both the directory containing the files to be compiled and the directory where the compiled files will be placed.
-cflags *compiler_flags* **/COMPILER =** *compiler_flags*	Specifies any C++ compiler options. Any header file specifications included here are used before the specifications included in the C project definition. For more information about these options, see "Extended External Properties" on page 195.
-lflags *linking_flags* */LINKING = linking_flags*	Specifies any linking flags. Any files included here are linked before files specified in the extended properties of the C project definition. For more information about specifying linking flags in the C project, see "Extended External Properties" on page 195.
-fm *memory_flags* **/MEMORY =** *memory_flags*	Specifies the space to use for the memory manager. See *A Guide to the iPlanet UDS Workshops* for information.
-fl *logger_flags* **/LOGGER =** *logger_flags*	Specifies the logger flags to use for the command. See *A Guide to the iPlanet UDS Workshops* for information.
-cleanup **/CLEANUP**	Deletes all the files except for the newly compiled shared libraries.

If the C functions are defined without prototypes, you need to include a special flag on the `fcompile` command. In this case, the you must add the following flag to define the `#define` constant `FORTE_NO_PROTOTYPES:`

```
-cflags '-DFORTE_NO_PROTOTYPES'
```

By defining this constant, the C++ wrapper code is compiled conditionally so that it will call the C functions correctly when they are not defined with prototypes.

For example, if you want to integrate a C function named “square” that takes an integer argument and returns an integer, you have to know whether the function is defined with or without prototypes, as shown in the following table:

C function with prototype (ANSI C)	C function without prototype
int square(int a)	int square(a)
{	int a;
return a * a;	{
}	return a * a;
	}

3. Copy the shared library, which you generated using `fcompile`, to the appropriate distribution directory in the following path:

`FORTE_ROOT``/`**`appdist`**`/`*environment_id/distribution_id/cl#/platform/component_id*

Directory name	Description
environment_id	First 8 characters of the environment name where you want your C project shared library to be installed.
distribution_id	8 character name derived from the C project name.
cl#	Compatibility level for this project, as specified in the C project definition.
platform	Architecture name for the platform where this shared library will be installed, for example VAX_VMS.
component_id	The *libraryname* value in the C project definition or the first 8 characters of the project name, if libraryname is not specified.

Install C Project Shared Libraries

Using the Environment Console or Escript, install the C project library distribution.

➤ **To install the C project library distribution**

1. In the Environment Console, choose the File > Load Distribution.
2. In the Load Distribution dialog, select the node on which you made the distribution for this library, then select the library distribution.
3. Choose the View > Application Outline command.
4. Select the library distribution that you just loaded, then choose the Component > Install.

You can also use Escript commands to install a C project library, as shown:

```
escript> LoadDistrib myCLibrary cl0
escript> Install
```

iPlanet UDS takes the shared libraries from the distribution library and installs the shared libraries on the appropriate nodes, according to the configuration you specified when you partitioned the C project.

For more information about installing libraries in iPlanet UDS, see *iPlanet UDS System Management Guide*.

Updating C Projects

➤ **To update an existing C project**

1. Update the C project definition in a file.
2. Check out all the components for this project using Fscript commands or the Project Workshop (see *Fscript Reference Guide* or *A Guide to the iPlanet UDS Workshops*).
3. Import the new or updated C project definition using Fscript commands or the Project Workshop (see *Fscript Reference Guide* or *A Guide to the iPlanet UDS Workshops*).

4. Check in the C project components by integrating your workspace using Fscript commands or the Project Workshop (see *Fscript Reference Guide* or *A Guide to the iPlanet UDS Workshops*).
5. Make a distribution, then compile, link, and install the project shared library, as described in this chapter.
6. If the iPlanet UDS partition that loads the C project shared library is running, shut it down and restart it to ensure that it loads the new C project shared library.

Making Installed C Projects Known to Other Repositories

After the shared libraries for your C project are installed in your environment, you can use the C project within your TOOL code to access the C functions.

If you also want other repositories to use this C project with its installed shared libraries, you must import the .pex file that was produced when the distribution was made. You can find this file on the nodes where the C project shared library was installed, in the same directory as the C project shared library. You can also find this file in the original distribution directory where the generated files were placed while making the distribution.

Chapter 11

Writing TOOL Code That Uses C Functions

This chapter explains how to include C functions in your TOOL application.

About Writing TOOL Code That Uses C Functions

Before you can write TOOL applications that use C functions, your repository must contain C projects that map to the C functions you want to use. These C projects must be installed and available as shared libraries on the appropriate nodes. These steps are explained in "About Making C Functions Available to iPlanet UDS Applications" on page 165.

➤ **To call a C function from within a TOOL application**

1. Add the C project for the C functions as a supplier project to your TOOL project.
2. For a distributed application, define a service object that will reside on the same node as the C function.
3. Write the TOOL application that uses the C functions.
4. Test your application.
5. Partition your application.
6. Deploy your application.

The remainder of this chapter describes these steps in details.

Examples

The examples in this section are based on the sample program DMathTm. DMathTm shows how you can write a C project definition that makes some ANSI C standard library functions available to TOOL programs.

For information about DMathTm, see Appendix A, "iPlanet UDS Example Applications."

Add the C Project as the Supplier Project

For example, the following lines indicate the supplier classes for a TOOL project that uses the C project, which are FrameWork and the C project DistMathAndTimeProject:

```
includes FrameWork;
includes DistMathAndTimeProject;
```

Project: TestDistMathAndTimeProject

You can also include your C project as a supplier project from within the Project Workshop. For more information about using the Project Workshop, see *A Guide to the iPlanet UDS Workshops*.

For a Distributed Application, Define a Service Object

iPlanet UDS has restrictions that affect how you should write an application that interacts with C functions:

- iPlanet UDS does not pass structs and unions across partitions.

  In your application, you can define a service object that resides on the same partition as the C functions. Within this service object, you can invoke all the C functions that use data structures, unions, and enums.

  This service object can receive objects from other partitions that contain data to be passed to the C functions. This service object can then copy the data from the objects into enums, structs, and unions and pass this data to the C functions.

  The service object can also copy information from the enums, structs, and unions passed back by the C functions into attributes of TOOL objects. Then, you can pass this information to other partitions using these objects.

- If a TOOL project uses a class that creates an instance of a restricted C class, then that TOOL class must also be restricted and reside on the same nodes as the restricted C class.

  Within a service object, you can copy the information from an object of a restricted C class to an object of a non-restricted TOOL class. You can then pass the non-restricted TOOL object to partitions that do not reside on the nodes where the restricted C class must reside, for example, a client partition.

- If an iPlanet UDS client partition makes a direct call to a C function, using a method in a C project, the client partition freezes while it waits for the C function to finish processing.

You can limit the effects of these restrictions by invoking the C functions within a service object that resides on the same partition as the C functions.

If your C functions are not restricted and they use only simple data types in their parameters, you can declare service objects of your C classes within your C project and access the C functions directly through these service objects. For more information about declaring service objects in C projects, see "Service Objects" on page 169.

The following example shows how you can define a service object of the AccessToMathAndTimeClass called accessToMathAndTime:

```
service accessToMathAndTime : AccessToMathAndTimeClass;
```

Project: TestDistMathAndTimeProject

Write the TOOL Application

Now you can write the TOOL application that uses features provided by the C functions.

Instantiate an Object for the C Class You Want to Use

Within your application code, you must instantiate the C class that has the methods for the C functions before you can invoke these methods.

The following example shows how to instantiate the C class `TimeFunctions`:

```
  tf : timeFunctions = new;
```

Project: TestDistMathAndTimeProject • **Class:** AccessToMathAndTimeClass
• **Method:** Test

Use the Methods of the C Class

After you have instantiated the C class, you can invoke methods on the object of that class. These methods invoke the underlying C functions.

The following example shows how to invoke the C functions time and localtime using the time and localtime methods provided by the C project:

```
-- Instantiate the timeFunctions class in the C project
  tf : timeFunctions = new;
-- Declare variables
  time_struct_ptr : pointer to tmstruct;
  t : time_t;
-- Get the current calendar time
  t = tf.time(nil);
-- Convert current calendar time to local time
  time_struct_ptr = tf.localtime(&t);
```

Project: TestDistMathAndTimeProject • **Class:** AccessToMathAndTimeClass • **Method:** Test

Map C Function Parameters to TOOL Method Parameters

When the C function receives or returns a value, the TOOL method must be able to correctly interpret this data. You might need to refer to documentation for the original C functions to understand exactly what data your TOOL method needs to provide or expect.

For more information about mapping C function parameters to TOOL method parameters, see "Mapping C Function Parameters in TOOL Methods" on page 239.

Include Error Handling

Most C functions return a return value when the function ends and passes control back to the TOOL application. What the return value means depends on how the C function defines it. Each C function should provide information describing how it returns error information to applications that invoke the function.

Test Your Application

You can test your TOOL application in the Project Workshop using the Run Distributed command. However, the shared libraries for your C projects that your application uses must be installed and available on the appropriate nodes. These steps are explained in "About Making C Functions Available to iPlanet UDS Applications" on page 165.

For information about using the Project Workshop, see *A Guide to the iPlanet UDS Workshops*.

Troubleshooting

This section includes a few tips that can help you troubleshoot iPlanet UDS applications that use functions provided by external C libraries.

Unexpected Failures

You should at some point deploy your application to test it is so that you can determine whether the default runtime stack size is sufficient for your application. If the external C library uses recursion, a large amount of local data in functions, or has a deep call graph, you should consider changing the FORTE_STACK_SIZE environment variable to at least 100000, and perhaps more, depending on the results of testing. If a partition of your application fails for no obvious reason, try increasing the FORTE_STACK_SIZE.

iPlanet UDS System Management Guide describes FORTE_STACK_SIZE more fully.

Unable to Locate the 3GL Supplier Library

iPlanet UDS automatically generates a C project .pex file when it makes a library distribution. This .pex file contains information about the distribution id and path where the library distribution will be installed. This information is also automatically stored as part of the C project in the repository from which you made the library distribution.

You might need to import the generated .pex file in the following situations:

- You want to use the wrappered C library as a supplier project in a repository other than the original repository from which you made the distribution. In this case, you need to import the .pex file into the second repository.
- The distribution information stored in the C project in the original repository was lost. For example, someone might have reimported the original hand-written C project file after the library distribution for the C project was made.

Partition Your Application

➤ **To partition your DCE client application**

1. Open the Project Workshop for the main project for your application.
2. In the Project Workshop, choose Run > Partition.

   iPlanet UDS displays the default partitioning for the library in the Partition Workshop.

You can also use the Fscript command `Partition`.

For information about partitioning your application, see *A Guide to the iPlanet UDS Workshops*.

Deploy the Application

The steps for making a distribution, compiling and linking shared libraries and installing your application are the same as for other TOOL applications.

For detailed information about performing these steps for TOOL applications, see *A Guide to the iPlanet UDS Workshops*.

NOTE In VMS, if the iPlanet UDS partition that invokes methods in the C shared library is also compiled, then you must define a logical that indicates the location of the installed C shared library. The following example shows how to define a logical for a C shared library named `MYLIB.EXE`:

```
define /TABLE = FORTE_GBLTABLE_version_number mylib FORTE_ROOT:[USERAPP.MYLIB.CL0]MYLIB.EXE
```

The *version_number* is the currently installed release of iPlanet UDS, for example V30E0.

Chapter 12

TOOL Statements for Defining C Projects

iPlanet UDS provides special versions of some TOOL statements to let you define C projects:

- `begin c` statement
- `class` statement

The syntax for these statements and usage information for these statements are included in the following pages. These sections also include restrictions for these statements.

begin c

The `begin c` statement defines a C project.

Syntax

begin c *project_name*;
[**includes** *supplier_project_name***;**] . . .
has property restricted = {TRUE | FALSE}
definition_list
[**has property** {*project_property;*}...]
definition_list
end [*project_name*];

project_property is:

[compatibilitylevel = *integer_constant*]
[multithreaded = {**TRUE** | **FALSE**}]
[libraryname = *string_constant*]
[extended = ([, **externalincludedirectories=**'*directories*']
[, **externalincludefiles =** '*include_files*']
[, **externalobjectfiles =** '*object_files*']
[, **externalstaticlibs =** '*static_libraries*']
[, **externalsharedlibs =** '*shared_libraries*'])]

Description

The `begin c` statement lets you define a C project in a file. To import the project definition from the file into your development repository, you can use the `ImportPlan` command in Fscript or the Import command in the Repository Workshop.

You can have more than one `begin c` statement for the same project. These can be in the same file or in different files. If the project already exists, iPlanet UDS simply adds the new definitions to the existing project.

If there is more than one definition for the same project component (for example, more than one definition for the same class name), iPlanet UDS uses the last definition.

Project Name

The value of *project_name* can be any legal iPlanet UDS name. If the name is unique, iPlanet UDS creates a new project. If the name already exists, it must be for a C project (not a TOOL project). When you specify an existing C project name, iPlanet UDS adds the definitions in the `begin c` statement to the existing project.

Includes Clause

The includes clause lets you provide a list of a supplier projects for the C project you are defining. Only a C project can be a supplier project for another C project. If your project needs to access definitions or services defined in another C project, you must add an includes clause that specifies that C project as a supplier project.

Name scope in a C project If you include a C project as a supplier project to a main C project, you must make sure that all the names, including those of defined unions and structs, are unique within the scope of the main C project and all its supplier C projects. Because C is case-sensitive, you can have names that are unique only in the way they are capitalized defined in different C projects. However, you cannot define names that differ only by case within a C project, because TOOL itself is *not* case-sensitive, and will not recognize the difference between the two names.

Definition List

The definition list is comprised of class statements and constant statements. For information about class statements, see "class" on page 198. For information about constant statements, see *TOOL Reference Guide*.

Has Property Clause

The `has property` clause lets you specify the C project properties. This section provides a brief description of each of the C project properties.

You can have more than one has property clause within the `begin c` statement. If you set the same property more than once, iPlanet UDS uses the last setting.

You can use the following properties with the has property clause:

- `restricted` property
- `compatibilitylevel` property
- `multithreaded` property
- `libraryname` property
- `extended external` properties

These properties are described in detail in this section.

restricted Property

The `restricted` property specifies whether or not your project is restricted. A project is defined as restricted if it can run only on particular hardware or software. The default is **restricted=**FALSE, which means that the product can run everywhere in your environment.

compatibilitylevel Property

The `compatibilitylevel` property lets you specify the compatibility level for the project. In general, if you plan to release a new version of your project, you should raise its compatibility level. Raising the compatibility level allows you to install and run the new release of the project in the same environments where older versions of the project are installed. The default is **compatibilitylevel**=0.

If you only change the implementation of the C functions themselves, you do not have to alter the compatibility level. For information about when you need to change the compatibility level of a C project, see *A Guide to the iPlanet UDS Workshops*.

multithreaded Property

The `multithreaded` property specifies whether your project is thread-safe. If multithreaded is set to FALSE, all other tasks are suspended when a task accesses a method in a C project until that function finishes. Thus, if your C functions cannot be executed simultaneously, you can still use them in a multi-tasking environment.

If you want your C functions to be executed simultaneously (**multithreaded=**TRUE), then you need to ensure that one or more instances of all the C functions defined in this project can run simultaneously without overwriting common memory. Also, your C functions must be signal tolerant, which means that signals generated by a running C function should not affect or be affected by signals generated by other tasks.

The default is **multithreaded=**TRUE.

libraryname property

The `libraryname` property specifies the name of the iPlanet UDS object module. The libraryname value must be unique and up to 8 characters long. This name must be unique so that once compiled, the shared library file will not conflict with the user object module file. If you omit this property, the default is the first 8 letters of the project name.

Extended External Properties

iPlanet UDS supports three extended external properties for C projects:

Extended Properties	Description
externalincludedirectories=*'directories'*	Specifies the directories containing header files to be included.
externalincludefiles=*'include_files'*	Specifies the header files to be included.
externalobjectfiles = *'object_files'*	Specifies object files to be linked into the project's shared library.
externalstaticlibs = *'static_libraries'*	Specifies static libraries (archives) to be linked into the project's shared library.
externalsharedlibs = *'shared_libraries'*	Specifies shared libraries to be linked into the project's shared library.

directories, *include_files*, *object_files*, *static_libraries*, and *shared_libraries* are all quoted string values. *directories* contains a list of directories separated by spaces. *include_files*, *object_files*, *static_libraries*, and *shared_libraries* contain a list of file paths and names separated by spaces. When you specify these values, follow these rules:

- Specify directories and file names in iPlanet UDS portable form and use absolute paths.
- Do not specify file extensions.
- You can use environment variables in the directory paths. These variables are resolved on the machine where the project's shared library is built. If the environment variables contain a path specification, use the "%" form of variable expansion.

  For example:

```
externalsharedlibs = '%{FORTE_ROOT}/mylibs/mylib
%{USRLIB}/libsocket'
```

You must specify the `externalincludedirectories` and `externalincludefiles` properties to define where iPlanet UDS should look for the header files it needs when it compiles the C++ wrapper code. If you do not set `externalincludedirectories` in the project definition file, you need to specify this information on the `fcompile` command with the -cflags or /COMPILER flags, usually using the -I C++ compiler flag. However, in this case, you cannot use the auto-compile feature when making a distribution.

The `externalobjectfiles`, `externalstaticlibs`, and `externalsharedlibs` properties specify the files that will be included as linking options when the compiled shared libraries are linked, as explained in "Compile and Link Shared Libraries" on page 177. iPlanet UDS links the specified files automatically and adds the files specified on these properties to the linking flags (-lflags or /LINKING) of the `fcompile` command in the following order:

Linking Order	Linked Files
1	Object files. Object files are linked in the order they are specified for the `externalobjectfiles` property.
2	Static libraries, which are sometimes archive files. Static libraries are linked in the order they are specified for the `externalstaticlibs` property.
3	Shared libraries. Shared libraries are linked in the order they are specified for the `externalsharedlibs` property.

If you explicitly specify some linking options on the `fcompile` command, iPlanet UDS adds the linking options from these extended properties to the end of the list of linking options.

For example, you might specify the following extended properties for your C project:

```
externalsharedlibs = 'standard math'
```

You might then specify the following linking options on the `fcompile` command:

```
fcompile -l '%{TMP}/time %{WORKING}/private_lib'
```

iPlanet UDS links the compiled shared libraries as though you had specified the following linking options on the `fcompile` command:

```
fcompile -l '%{TMP}/time %{WORKING}/private_lib standard math'
```

If you are using the auto-compile option when making a distribution, then iPlanet UDS uses the files specified here as the linking options when iPlanet UDS compiles and links the shared libraries, as described in "Making the Distribution with Auto-Compile and Auto-Install" on page 174.

For information about the `fcompile` command, see "Compile and Link Shared Libraries" on page 177.

class

Use the `class` statement to create a C class. The form of the `class` statement used in a C project is a restricted version of the statement found in the *TOOL Reference Guide*.

Syntax

class *class_name*
[*component_definitions*]...
[**has property** [**distributed** = (**allow** = {**on** | **off**} [, **override** = {**on** | **off**}]
[, **default** = {**on** | **off**});]...]

Description

You can define the following components for a class:

- methods
- constants

Methods

The syntax for including a method in a C project class definition is the same as for any other method in TOOL. Refer to *TOOL Reference Guide* for the syntax of defining methods.

NOTE You need to map the parameters and return values for the C functions to equivalent TOOL data types. For more information about this mapping, see "Mapping Simple C Data Types to TOOL Data Types" on page 201 and "Mapping Derived C Data Types to TOOL Data Types" on page 203.

Chapter 13

Using C Data Types in TOOL

With iPlanet UDS, you can write applications that interact using several industry-standard products and protocols, as described in this manual, *Integrating with External Systems*.

In iPlanet UDS, you can use several standard C data types to pass parameters between iPlanet UDS TOOL methods and certain types of external applications.

This chapter discusses the following topics:

- using C data types in TOOL methods
- dynamically allocating and deallocating storage for C data types in TOOL
- mapping C function parameters to TOOL method parameters

Using C Data Types in TOOL Methods

This chapter describes how C data types map to iPlanet UDS TOOL data types, and how to use these data types. This chapter also explains how to manage dynamic memory allocation and deallocation for C data types.

You can write TOOL applications that interact with several external object services that provide C language application program interfaces (APIs), including ObjectBroker, DCE, OLE 2, and C functions. This chapter uses the term *C functions* to refer to both C language APIs and C functions.

Derived data types To write these TOOL applications, you must first describe each C function to iPlanet UDS by writing a project definition in a file. This project definition maps the C functions and data types to TOOL methods and data types. Fortunately, TOOL supports data types that map directly to most of the data types used by these services. However, you cannot pass objects to C functions. Therefore, TOOL provides simple data types and *derived* C data types that you can use to

transfer an object's data to and from a C function. Derived data types are data types that are defined using other data types, like enums and structs. These mappings are explained in "Mapping Derived C Data Types to TOOL Data Types" on page 203.

As in C, when you declare variables with derived data types, you need to consider whether you want to use local variables on the runtime stack, or whether you want to allocate memory dynamically. In general, you can use local variables if you do not need the data held by the variable beyond the end of the method. However, if you want to reference the variable outside of the current method, then you need to dynamically allocate the memory, then later free the memory when your application no longer needs the variable. Using dynamic memory allocation is explained in "Managing Memory for C-style Arrays and Data Structures" on page 232.

To determine how to map the C function parameters to the parameters of the corresponding TOOL method, you must understand exactly what the C function intends to pass and why, and whether the calling method will later want to reference the value of a parameter. This topic is discussed in detail in "Mapping C Function Parameters in TOOL Methods" on page 239.

For examples of executable code that demonstrates using C data types in TOOL, see the example programs in the following files:

- `AllCTypes.pex`
- `MathTime.pex`

These examples are described in Appendix A, "iPlanet UDS Example Applications."

General Guidelines

The TOOL implementations of C data types generally behave like the corresponding data types in ANSI C. These data types also generally follow the same syntax rules as for other TOOL data types. Be aware of the following guidelines:

- Declare variables of these data types as you do for TOOL, using the following syntax:

  variable_name **:** *data_type;*

- Do not instantiate any variables for C data types with the `new` keyword.

- Casting for most C data types works the same for C data types as for simple data types in TOOL. For information about casting in TOOL, see the *TOOL Reference Guide*. Any differences for the derived data types are documented in the description for the specific data type.

Mapping Simple C Data Types to TOOL Data Types

The parameter types for the C functions and their corresponding TOOL types are shown in the table below:

C Data Type	TOOL Data Type
int	int
long	long
short	short
float	float
double	double
unsigned int	uint
unsigned long int	ulong
unsigned short int	ushort
char	char
char	i1
char *	pointer to char
char *	string (limited usage, see description below)
unsigned char	ui1

The numeric TOOL data types directly correspond to the listed C data types; therefore, the ranges of these data types depends on the machine you are using at runtime.

iPlanet UDS provides `uint` to map to C's unsigned int and `ulong` to map to C's unsigned long int. The following table describes these data types:

TOOL Data Type	Description
uint	0 to at least +65535
ulong	0 to at least +4,294,967,295

For descriptions of the other TOOL simple data types, see the *TOOL Reference Guide*.

In general, to map a char * variable to a TOOL variable, use a pointer to char. You can use a string to map to a char * C parameter *only* in the following situations:

- the location of the data is not important

  TOOL string variables are declared on the runtime stack for a method in memory managed by TOOL. If you call a C function and pass a string parameter, the string data remains in the same location while the C function is running. However, the string data might move between C function calls.

  If you choose to use pass a string parameter to a C function, the C function should not store the address of a string in global memory or in dynamically allocated memory for later use by another C function. The address of the string might change when iPlanet UDS reorganizes its managed memory between the exit of the first C function and the invocation of the second C function.

- the char * is not part of an enum, struct, or union

  TOOL does not allow string variables to be members of enums, structs, or unions.

Use non-portable data types Although you normally want to use portable data types in your TOOL code, you should use the non-portable types recommended in the table above to map TOOL method parameters to the parameters of your C functions. Otherwise, your TOOL application might use a different data type than your C function. For example, if your C function runs on a PC, then an integer parameter defined in the C function with the `int` key word is 2 bytes long. However, if you map this parameter in a TOOL method using the `integer` key word, then TOOL will try to pass a 4-byte integer to this C function.

The following TOOL types do not correspond directly to C types in a machine-independent way: ui2, ui4, i2, i4, and integer. Avoid using these types when you call a method that maps to a C function.

Mapping Derived C Data Types to TOOL Data Types

Derived C data types are constructed data types that you can define using simple data types. Variables declared using derived data types identify areas of memory.

Derived C data types and their corresponding TOOL data types are shown in the table below:

C Data Type	TOOL Data Type	Description
array	C-style array	A set of a predefined size that contains related values of the same data type.
enum	enum	A finite set of integers that declares identifiers for each value defined for the set.
* (pointer)	pointer to *data_type*	Pointer to a specific type of data.
void * (pointer)	pointer	Generic pointer
struct	struct	A defined set of C variables of various data types.
typedef	typedef	An identifier associated with a specific data type
union	union	A set of variables whose values can be stored in the exact same memory space; only one of these variables can be stored at a time.

The following sections explain how to declare these TOOL data types. For information about how to map TOOL data types for supported C functions, see the specific chapter for that interface.

Restrictions

You cannot pass derived data types, except enums:

- when passing data between partitions

  iPlanet UDS does not support passing derived data types, except enums, between iPlanet UDS partitions.

- as parameters of events

  Although you cannot pass derived data types, other than enums, as parameters of events, you can pass pointers that reference dynamically-allocated C-style arrays or structs as parameters of events, as well as pointers to other dynamically-allocated data types. This dynamically-allocated data must still exist by the time a registered event handler tries to handle the event.

  Note, however, that you cannot pass events that have pointers as parameters between partitions. In other words, if you post an event with a pointer as a parameter and try to register for and handle the event in another partition, you get an error.

  For more information about dynamically allocating memory for these C-style arrays and structs, see "Managing Memory for C-style Arrays and Data Structures" on page 232.

- as parameters of methods that are started using a `START TASK` statement

  You cannot pass derived data types, other than enums, as parameters of methods that are started using a `START TASK` statement. You can, however, pass pointers that reference dynamically allocated C-style arrays or structs as parameters for these methods. For more information about dynamically allocating memory for these data types, see "Managing Memory for C-style Arrays and Data Structures" on page 232.

All derived data types, except pointers and C-style arrays, can only be defined at the project level. You cannot define enum, struct, typedef, or union data types within methods.

C-style Arrays

A C-style array is a set of a predefined size that contains related values of the same data type. *A C-style array is similar to arrays used in ANSI C.*

CAUTION You cannot pass C-style arrays between partitions, as parameters of events, or as parameters of methods that are started using a `START TASK` statement.

Differences Between Array Objects and C-style Arrays

TOOL objects of the `Array` and `LargeArray` classes are fundamentally different from C-style arrays. The following table compares them.

	Array or LargeArray Object	C-style Array
Description	An object that stores and manipulates a collection of objects using methods of the array class.	A group of data storage locations that have the same name and are distinguished from each other by an index.
What it contains	Objects. It cannot contain simple or derived data types.	Simple and derived data types. It cannot contain objects.
How to allocate storage for it	Instantiate it using the **new** keyword to allocate memory.	Declare a C-style array on the runtime stack, or dynamically allocate memory for a C-style array using calloc or malloc.
Can be the return value for a TOOL method	Yes.	No.

Declaring Arrays on the Runtime Stack

The following table shows the syntax for mapping an array for a C function to a TOOL C-style array. iPlanet UDS supports two variations of the syntax for declaring a C-style array.

This table shows a simplified syntax diagram, which includes all possible elements of the array syntax. The brackets "[" and "]" represent characters that are part of the syntax.

C Syntax	TOOL Syntax
data_type name [*size*] [*size*]...	**name : array** [*lower***..***upper*] [l*ower***..***upper*]... **of** *data_type***;**
	name : array [*lower***..***upper* **,** *lower***..***upper* ...] **of** *data_type***;**

In the C syntax:

- *data_type* is the name of the data type for all the elements of the array
- *name* is the variable name for the array
- *size* is the number of elements in the array

In the TOOL syntax:

- *name* is the variable name for the array
- *lower* and *upper* define the lower and upper bounds of the array, and their values must be integer constants

  A value for the *lower* bound is not required, but if you specify it, the value must be lower than the value of the corresponding *upper* bound.
- *data_type* is the name of the data type for all the elements of the C-style array

The following example illustrates declaring C-style arrays in TOOL using the syntax variations:

C Example	TOOL Example
`int myArray[5][3];`	`myArray : array[5][3] of int;`

Unless you specify the lower bound of the C-style array, the numbering of these array elements starts at 0, as in C.

If you specify a *lowe*r bound for the TOOL C-style array and pass this array to a C function, the C function still indexes this array starting at 0. The following example shows how the TOOL C-style array maps to an array in the C function:

C Example	TOOL Example
`int myArray[5];`	`myArray : array[5..10] of int;`

You can specify empty brackets, "[]", for a C-style array that is a parameter of a C function to indicate that this C-style array is of unknown size. TOOL manages this array as a pointer to the data type that the array contains. Therefore, certain error messages involving this C-style array might refer to a pointer instead of an array.

The following example shows how you can map an array in C to both variations of the C-style array syntax in TOOL:

C Example	TOOL Examples
`int myArray[5][3];`	`myArray : array[5][3] of int;` `myArray : array[5, 3] of int;`

Using the first variation of the syntax, you can specify arrays as shown in the following examples:

```
-- Declare a one-dimensional array with 10 integer elements numbered
-- 0 to 9
myarray1 : array [10] of int;
-- Declare a one-dimensional array with 10 char elements numbered
-- 1 to 10
myarray2 : array [1..10] of char;
-- Declare a two-dimensional array containing 5 arrays that contain
-- 8 float elements numbered 3 to 10
myarray3 : array [5][3..10] of float;
```

Using the second variation of the syntax, you can specify a multi-dimensional array as shown in the following example:

```
-- Declare a two-dimensional array containing 6 arrays of 8
-- integer elements
myarraya : array [6, 8] of int
-- Declare a two-dimensional array containing 5 arrays that
contain
-- 8 elements that are numbered 3 to 10
myarrayb : array [5, 3..10] of float
```

Declaring C-style Arrays Dynamically

When your application needs a C-style array to exist *after* the current method exits, you must dynamically allocate the memory for the array. To dynamically allocate the array, you declare a pointer to a C-style array, then allocate the needed storage for the array using the C functions **malloc** or **calloc**. For more information about dynamically allocating storage, see "Managing Memory for C-style Arrays and Data Structures" on page 232.

Converting C-style Arrays of Char to TextData Objects

iPlanet UDS TextData objects have many methods that are useful for working with strings, so you may want to convert C-style arrays of char to TextData objects.

To convert a C-style array of char to a TextData object, use the Concat method of the TextData class. The following example shows how you can convert a null-terminated string stored in a C-style array of char into a TextData object:

```
-- myCharPointer is a pointer to char that references a null-
-- terminated string. TOOL stores this string as a
-- C-style array[23] of char, with 22 characters and 1 NULL.

-- Declare and instantiate a TextData object, then copy the
-- C-style array of char into the TextData object.
OurTextObject : TextData = new;

-- Concat reads characters until it reaches the null terminator '\0'
-- and copies the characters as the value of StringObject.Value
OurTextObject.Concat(myCharPointer);
```

The following example shows how you can convert a value stored in a C-style array of char without a terminating '\0' value into a TextData object:

```
-- Set up an array of char, and initialize with some characters
MyArray : array[100] of char;
-- Put 15 characters of data into the first 15 positions of the
-- C-style array with no null characters
...
-- Copy the C-style array of char into the TextData object
NewTextObject : TextData = new;
NewTextObject.Concat(MyArray, 15);
```

Converting TextData Objects to C-style Array of Char

You cannot pass iPlanet UDS objects to C functions. Therefore, to pass the value of a TextData object as an input parameter to a C function, you can pass the string value of the Value attribute of the TextData object. However, if you want to pass a string to a C function and reference any changes that the C function might have made to the string, you need to load the Value attribute into a C-style array of char.

To load the characters in the Value attribute of a TextData object into a C-style array of char, use the ExtBytes method of the TextData class, as shown in the following example:

```
-- Declare an array[20] of char
MyArray : array[20] of char;

-- Declare and instantiate a TextData object that contains a
-- string value
myString : TextData = new(value = 'String of sample text.');

-- Copy the first 19 characters in MyString into MyArray
myString.ExtBytes(MyArray, 19);
```

Converting TOOL Strings to C-style Arrays of Char

You might need to convert TOOL string values to C-style arrays when:

- the location of the data is important to your application, because TOOL might move the string values when it reorganizes its managed memory
- you need to assign the string value as part of a struct or union
- you want the data to exist beyond the end of the current method

To convert a TOOL string to an array of char, use the **strdup** C function to allocate a space outside of the memory managed by the iPlanet UDS runtime system and copy the string value to this area of memory. For more information about strdup, see "strdup" on page 236.

When your program finishes using the converted array of char value, you need to explicitly free the memory referenced by the pointer. If you do not, you will reduce the amount of available memory. The following example shows you how to copy a string value to memory not managed by the iPlanet UDS runtime system, then free the pointer:

```
-- Declare a pointer to char
myCharPointer : pointer to char;
-- Convert string value to a C-style array of char values
myCharPointer = strdup('This is a string.');
-- Use the values
...
-- Explicitly free the memory used by the array of char values
free(myCharPointer);
```

Enumeration Data Types (enums)

Enumeration data types (enums) let you define a set of integer values with identifiers as elements of the set. Enums are useful for ensuring that the value assigned to a variable falls into an appropriate set of values.

CAUTION You can define enums within projects, classes, and structs. However, you cannot define enums within methods.

The following table shows how you can map an enum data type declaration between a C function and a corresponding TOOL method.

C Syntax	TOOL Syntax
enum *enum_name* {*item_name* = *constant* , *item_name* = *constant* ...};	**enum** *enum_name* *item_name* [= *constant*], [*item_name* [= *constant*], ...] **end** [**enum**];

The name *enum_name* becomes a data type name in the current name scope. The enumeration data type, *enum_name*, contains the names of the items, so you must reference the items using dot notation, as shown:

enum_name.*item_name*

Each named item has an integer value associated with it. By default, each item is numbered sequentially starting with zero, unless you explicitly define a value for one or more of the items. Two or more items can have the same value.

NOTE iPlanet UDS does not automatically convert integers to enum values, so you must explicitly cast an integer to an enumeration data type in your TOOL code. iPlanet UDS does not perform runtime checking to ensure that the integer value is legal for the data type.

The following example shows how you could map a C enum data type declaration to a TOOL enum data type declaration:

```
-- In the project, declare an enumeration data type that contains
-- three colorful identifiers numbered 0 to 2
enum color
   red,
   yellow,
   blue
end enum;

-- In a method, use the enum identifiers.
i: int;
i = color.yellow         -- i = 1
-- Declare a variable of enumeration data type color
rainbow : color;
-- Set rainbow to red using an integer value (color.red = 0)
MyInteger : int = 0;
-- Need to cast the integer value to color enum data type
rainbow = (color)(MyInteger);
```

The items in the `color` data type can only be accessed as `color.red`, `color.yellow`, and `color.blue`. For example, in the `color` data type, the value of `color.red` is `0`, the value of `color.yellow` is `1`, and the value of `color.blue` is `2`.

The following example shows you how to assign specific integer values to the items of the enumeration data type:

```
-- In the project, declare a enumeration data type and assign some
-- specific integer values
enum myEnum
   firstItem = 25,
   secondItem = 24,
   thirdItem,
   fifthItem = -6,
   sixthItem,
   seventhItem = 24
end;

-- In a method, assign the value of an enum identifier to another
-- variable.
MyInt1, MyInt2 : integer;
-- secondItem = 24; the value of thirdItem is one greater than
-- secondItem, so it is 25.
MyInt1 = myEnum.thirdItem;    -- MyInt1 = 25
-- fifthItem = -6; sixthItem is one greater than
-- fifthItem, or -5
```

```
MyInt2 = myEnum.sixthItem;     -- MyInt2 = -5
-- Declare a variable of enumeration data type myEnum
MyEnumData : myEnum;
```

Notice that type myEnum can have several items with the same value.

Pointers

iPlanet UDS provides two kinds of pointers: generic pointers and pointers to specific data types. iPlanet UDS does not support pointers to functions.

This section describes how to declare and cast pointers that can be passed to C functions. For information about using pointers when passing parameters, see "Mapping C Function Parameters in TOOL Methods" on page 239.

CAUTION You cannot pass pointers between partitions or as parameters of methods that are started using a START TASK statement.

You can pass pointers that reference dynamically allocated memory as parameters in situations that use asynchronous processing, such as posted events, multitasking, or calling external routines. However, you must ensure that the application frees the memory for these parameters only after all the event receivers and all the tasks are finished using the parameter values. You cannot pass these pointers between partitions.

Generic Pointers

Generic pointers store an address retrieved from a C function. This pointer is equivalent to a void * pointer in C.

The following table shows how you can map a C pointer to a generic TOOL pointer.

C Syntax	TOOL Syntax
void * *pointer_name;*	*pointer_name* **: pointer;**

You can use this type of pointer to store the addresses retrieved from C functions. Later, you can assign this value to the pointer parameter for another C function. This is particularly useful with C interfaces that pass opaque pointers.

CAUTION If you use a generic pointer in iPlanet UDS, iPlanet UDS cannot check that the value referenced by the pointer has the correct data type. Therefore, you must make sure that the data type of the value you assign to the generic pointer in your application is appropriate.

The following example shows how you can:

- retrieve values from a C function called GetStatHandle in a C function library (StatisticsLibrary class) by using a pointer
- pass the pointer to another C function associated with the method ReadStatHandle

c_stat_package is the name of a class that has GetStatHandle and ReadStatHandle as methods.

```
c_stat_package : StatisticsLibrary = new;
-- C interface
c_handle : pointer;
-- Call a C function, which returns a C pointer, and pass to
-- another C function.  The TOOL program never uses c_handle
-- directly.
c_handle = c_stat_package.GetStatHandle();
c_stat_package.ReadStatHandle(c_handle, "open");
```

Pointers to Specific Data Types

TOOL lets you define a pointer that references a specific data type. You can use pointers like these to ensure that the pointer references a data item of the correct type.

The following table shows how you can map a C pointer to a specific data type to a TOOL pointer.

C Syntax	TOOL Syntax
data_type * *pointer_name;*	*pointer_name* **: pointer to** *data_type***;**

pointer_name is the pointer variable name. *data_type* is the data type that the pointer references.

The following example shows how the TOOL pointer declaration syntax maps to C pointer syntax:

C Example	TOOL Example
`int * myPointer1;`	`myPointer1 : pointer to int;`
`float * * myPointer2;`	`myPointer2 : pointer to pointer to float;`

Dereferencing Pointers

To dereference a pointer to a specific data type, you can use the * and -> operators, as in C.

**variable_name* refers to the beginning of the simple data type, C-style array, or data structure that *variable_name* references.

pointer_variable_name->member_name refers to an element (*member_name*) of the data structure or union pointed to by *pointer_variable_name*. You can use the -> operator only when the pointer refers to a struct or union data type.

The following example shows how you can use pointers to pass a C-style array of integers to a C function called GetStatistics in the C function library (StatisticsLibrary class) and retrieve the results:

```
-- Call a C function that performs statistical analysis of the data,
-- returning the average, median, and so on. The input parameter is
-- an array of integers; this function returns a
-- pointer to a C data structure with following definition:
-- struct stat_results
--     average : float;
--     median : float;
-- end struct;

-- Instantiate the class with the C methods
c_stat_package : StatisticsLibrary = new;

-- Declare variables that pass values to and retrieve values from the C
-- functions
c_values : array [10] of int;
c_stats : pointer to stat_results;
-- Declare and assign values to the array pointed to by c_values
...
-- Invoke the C function
c_stats = c_stat_package.GetStatistics(c_values);

-- Dereference the returned pointer value to get the calculated values
ave, median : float;
ave = c_stats->average;
median = c_stats->median;
```

Address Operator (&)

iPlanet UDS provides an address operator (&) that determines the address in memory of the variable it precedes, just like in C. For example, &MyVariable indicates the address of the memory where the variable MyVariable is stored.

You can use the address operator to access the address of an existing value. You can then pass this address to method parameters that require a pointer value.

The following example shows how you can use the address operator (&) to pass the address of a value to a C function:

```
tf : timeFunctions = new;
time_struct_ptr : pointer to tmstruct;
t : int;
-- If you pass the C Library routine time() a NIL pointer,
-- it returns an int.
t = tf.time(nil);
-- You can then pass localtime() the address of this int.
time_struct_ptr = tf.localtime(&t);
```

To see this code fragment in context, see the iPlanet UDS example program dmathtm.pex.

You can pass the addresses of the following items as parameters:

- variables
- the beginnings of C-style arrays
- members of structs and unions
- attributes of objects

To pass the address of an attribute as a parameter, the class of the object must have the following properties set as shown:

```
distributed = (allow = off, override = off);
transactional = (allow = off, override = off);
monitored = (allow = off, override = off);
shared = (allow = off, override = off);
```

Duration of addresses The address of a TOOL variable is only valid for the period of a single C call; therefore, you cannot store the address of a TOOL variable in a C structure and later use the address.

NOTE You cannot use the address of any of the following data types:

- virtual attribute
- a row of a dynamic array
- an attribute of an object that has the possibility of being shared, distributed, transactional, or monitored

Pointer Constants

The only constant available for the pointer type is `NIL`. To assign a value to a pointer data item, you must specify another pointer of the same pointer type or invoke a method with a return value of the pointer type (see the *TOOL Reference Guide* for information on invoking methods).

Casting Pointers

TOOL automatically casts any pointer type to a generic pointer type. However, you must explicitly cast a pointer value when you assign it to another pointer type. The following example shows how you can cast pointers:

```
-- Define a generic pointer and a pointer to a specific type
myGenericPtr : pointer;
myIntPtr : pointer to integer;
-- The pointer to integer (myIntPtr) is automatically cast to
-- the generic pointer type
myGenericPtr = myIntPtr;
-- You must explicitly cast the generic pointer (myGenericPtr)
-- to the pointer to integer data type
myIntPtr = (pointer to integer)(myGenericPtr);
```

Struct Data Types

You can use the `struct` keyword to declare a data structure that contains one or more members. You can define structs within projects; however, you cannot define structs within methods.

CAUTION You cannot pass structs between partitions, as parameters of events, or as parameters of methods that are started using a `START TASK` statement.

The members in a structure can be of different data types, including C-style arrays and other data structures. However, the members in a structure cannot be strings or objects.

The following table shows how you can map a data structure declaration between a C function and a TOOL method:

C Syntax	TOOL Syntax
struct *structure_name* {	**struct** *structure_name*
data_type member_name;]...	[*member_name* : *data_type*;]...
};	[**has property opaque=TRUE;**]
	end [**struct**];

structure_name becomes a data type name in the current name scope. *member_name* is the name of a variable within the data structure. *data_type* is the data type for a variable in the data structure.

The statement `has property opaque=TRUE` indicates that a structure by this name is defined in a header file specified using the extended external attributes externalincludefiles and externalincludedirectories. For information about using opaque C data structures, see "Defining Opaque Structs" on page 224.

You cannot assign the values of members in a structure within the `struct` syntax. Instead, you must use individual assignment statements to assign the initial value for each member.

Data structures cannot contain TOOL string values. To store a string of character data in a structure, you must use an array of char or a pointer to char. For information about converting a string value to an array of char values, see "Converting TOOL Strings to C-style Arrays of Char" on page 210.

The following example shows how you could declare a struct data type named customer in both C and TOOL:

C Example	TOOL Example
`struct customer {`	`struct customer`
`  char LastName[20];`	`  LastName : array[20] of char;`
`  int IDNumber;`	`  IDNumber : int;`
`  char gender;`	`  gender : char;`
`};`	`end struct;`

A variable declared for a given struct data type actually contains the values of the data structure, not a pointer to the data structure.

Accessing Values in a Data Structure

To access the values in a data structure, use dot notation, as shown the following syntax:

structure_variable_name.member_name

structure_variable_name identifies a variable of a struct data type, and *member_name* is the name of a member in this data structure.

The following example shows how you would access the value of a member IDNumber in a data structure of data type customer declared in the previous example:

```
-- Declare a variable of the customer struct data type.
MyStruct : customer;
...
-- Assign values to the members of MyStruct.
...
```

```
x : int;
-- Reference the value of the IDNumber member using dot notation.
x = MyStruct.IDNumber;
```

If you reference a data structure using a pointer, you can use arrow notation to dereference the value of a member in the data structure. To dereference the values in this structure using arrow notation, use the following syntax:

pointer_to_structure_name **->** *member_name*

pointer_to_structure_name identifies a pointer to a data structure, and *member_name* is the name of a member in this data structure.

The following example shows how you would access the value of a member IDNumber in a data structure of data type customer declared in the previous example:

```
-- Declare a data structure of type customer.
MyStruct : customer;
-- Declare MyPointer and assign MyPointer the address of MyStruct
MyPointer : pointer to customer;
MyPointer = &MyStruct;
-- Assign values to the members of MyStruct
...
x : int;
-- Reference the value of the IDNumber member using arrow
notation.
x = MyPointer->IDNumber;
```

Alignment of Structs

The iPlanet UDS system assumes that structs defined by C applications use the default alignment defined by the C++ compilers supported by iPlanet UDS for each platform. In general, this default is to align the members of a struct at the natural boundary for the data type on that machine. However, for PC Windows, the default alignment uses a byte boundary.

If you intend to access a struct in a C application that is aligned differently than the default alignment expected by the iPlanet UDS system, you need to define iPlanet UDS structs to compensate for this difference between what iPlanet UDS expects and what the C structure provides. However, be aware that this mapping can be complex and is machine-dependent.

Defining Structs within Structs

To define a struct as part of another struct, you can declare structs within structs using the following syntax.

struct *outer_struct*
 [*list_of_members*]
 member_name **: struct** *inner_struct*
  [*list_of_members*]
 end [**struct**];
 [*additional_members*]
end [**struct**];

Structs that are declared as members within another data type exist only as part of the outer data type. You cannot reference that data type except within data structures of the outer data type.

The following example shows how you can define a struct data type containing structs as members:

```
-- Define a struct data type containing two other struct data types
struct s1
  s1_a1 : int;
  s1_a2 : struct s2
      s2_a1 : float;
      s2_a2 : struct s3
          s3_a1 : float;
          s3_a2 : array[10] of int;
      end;
  end;
end;
```

Alternatively, you can also define a struct for a project, then use this struct to define a member of another struct, as in the syntax shown here:

struct *inner_struct*
 [*list_of_members*]
end [**struct**];

struct *outer_struct*
 [*list_of_members*]
  member_name : *inner_struct***;**
  [*additional_members*]
end [**struct**];

Defining Opaque Structs

iPlanet UDS can use the definition of a struct that is defined in a C header file to generate the necessary C++ wrapper code. iPlanet UDS defines the struct as an *opaque struct*, which means that you can use TOOL to allocate storage for the structure and define pointers for this structure, but you cannot reference specific members of the structure unless you have defined them explicitly in the TOOL struct definition.

You might want to define opaque structs in the following cases:

- a struct is defined as opaque in an API, for example the LDIR type struct that is used but not defined in <dirent.h> for functions such as opendir
- the definition of a struct is system dependent, for example, the FILE type struct that is defined in <stdio.h>
- a struct is very complicated, and you only need to pass a variable of this struct type to a C function, but you will never examine the members of the struct

You can use the following syntax to define an opaque struct:

struct *struct_name*
 [*member_name* **:** *data_type***;**]
 has property opaque = TRUE;
end [**struct**]**;**

struct_name must exactly match, including case, the name of the struct in the C header file. *struct_name* becomes a data type name in the current name scope. The struct name must be unique within the scope of the C project.

member_name is the name of a variable within the data structure. *member_name* must exactly match, including case, the name of a member in the struct in the C header file. You only need to include member names if you want to access the members in your TOOL code.

data_type is the data type for a variable in the data structure, and must map to the data type of the associated member in the C header file, as described in this chapter.

Within your external project definition file, you must specify extended properties, `externalincludefiles` and `externalincludedirectories`, that identify the names and locations of the C header files that contain the structs defined in this project definition file. Each header file must be self-contained, that is, each header file must define or include all definitions required for that header file.

The following table shows how you can map a struct definition in a C header file as an opaque struct in an external project definition file. The definition of the struct in the C header file is shown here to illustrate the mapping to TOOL; however, in many cases, documentation about the definition of the struct might not be available.

C Header File Example (MyFile.h)	TOOL Example

`struct MyCStruct`
`{  int member1;`
`  char *Member2;`
`  long member3;`
`  short *MEMBER4;`
`};`
	`struct MyCStruct`
	`  has property opaque=TRUE;`
	`end;`
	
	`has property`
	
	`extended=(externalincludefiles='MyFile.h',`
	`  externalincludedirectories=`
	`  '%{MY_HEADER_FILES}');`

This example shows how you can use the MyCStruct struct defined in the MyFile.h C header file to define a struct that you can use in TOOL. In this case, the contents of the struct MyCStruct is unknown to TOOL.

You must define the name and location of the header file containing this structure using two extended external properties of the project definition: `externalincludefiles` and `externalincludedirectories`.

You can declare a variable of this struct type on the runtime stack, allocate storage for a struct of this type, and define a pointer of this struct type. However, because this struct is opaque to TOOL, the only thing you can do with the variable of this struct type is pass this struct to a mapping method for a C function that requires this struct type as input.

If you want to define an opaque structure that has some members visible to TOOL, you can define a struct as shown in the following example:

C Header File Example (MyFile.h)	TOOL Example

`struct MyCStruct`
`{  int member1;`
`  char *Member2;`
`  long member3;`
`  short *MEMBER4;`
`};`
	`struct MyCStruct`
	`  member1 : int;`
	`  Member2 : pointer to char;`
	`  has property opaque=TRUE;`
	`end;`
	
	`has property`
	
	`extended=(externalincludefiles='MyFile.h',`
	`  externalincludedirectories=`
	`  '%{MY_HEADER_FILES}');`

This example shows how you can use the MyCStruct struct defined in the MyFile.h C header file to define a struct that you can use in TOOL. In this case, you have also explicitly specified two members of the struct, member1 and Member2, in your project definition file; you can access these members using your TOOL code.

You must define the name and location of the header file containing this structure using two extended external properties of the project definition: `externalincludefiles` and `externalincludedirectories`.

If you do not set `externalincludedirectories` in the project definition file, you need to specify this information on the `fcompile` command with the -cflags or /COMPILER flags, usually using the -I C++ compiler flag. However, in this case, you cannot use the auto-compile feature when making a distribution.

For information about defining the externalincludefiles and externalincludedirectories extended attributes in your project definition and about the syntax of the `fcompile` command, see the information for the specific external system.

Determining the Name Scope of Structs

When you define a derived data type, TOOL adds the type name to the current scope, so that the current scope contains the type declaration.

For example, if a struct data type definition in a project, then the name of the struct data type becomes a type name within that project. Similarly, if a struct data type definition is part of another struct data type definition, as in the following example, then the name of the inner structure is part of the outer structure's name scope:

```
-- Define a struct data type

struct outerstruct
-- Declare another struct data type within outerstruct
   a : struct innerStruct
          b : integer;
       end struct;
end struct;
```

In this example, innerStruct is considered a component of outerStruct. To refer to innerStruct you must fully qualify it as outerStruct.innerStruct.

For more information about name scope in TOOL, see *TOOL Reference Guide*.

Typedef Data Types

The `typedef` key word lets you define synonyms for specific data types, just like you can in C.

CAUTION You cannot pass typedefs between partitions, as parameters of events, or as parameters of methods that are started using a START TASK statement.

You can only declare `typedef` data types in projects. You cannot declare typedefs within methods.

The following table shows how you can map a typedef declaration between a C function and a corresponding TOOL method.

C Syntax	TOOL Syntax
typedef *data_type type_name;*	**typedef** *type_name* **:** *data_type* **;**

type_name is the name that can be used as the name of a data type within the current name scope. *data_type* is the actual data type or data type definition.

The following example shows how you can map a typedef declaration between C and TOOL:

C Example	TOOL Example
`typedef char LastName[15];`	`typedef LastName : array[15] of char;`

The following example shows how using the typedef data type can simplify declaring variables of certain types:

```
-- Define a data type for an integer array
typedef intArray : array [1..10] of integer;
myFirstArray : intArray;
```

NOTE Typedefs exist in TOOL code only until you compile the code. Compiling the TOOL code includes compiling a TOOL project or importing a project definition for an external project. When you compile your TOOL code, any variables or derived data types that you defined with typedefs are redefined using the actual data type.

After you compile your project:

- Variables defined in your methods using typedef data types are now defined using the actual data type definition.
- Error messages refer to the actual data type, not to the typedef name.
- Log file information refers to the actual data type, not to the typedef name.
- When you debug your project, the data types of your variables will always be the actual data type, not the to typedef name.

Union Data Types

Unions are data types that define a set of alternative values, or members, that can be stored in the same space in memory. Only one member can be assigned a valid value at a time. TOOL supports only non–discriminated unions, like those in C.

CAUTION You can define unions within projects; however, you cannot define unions within methods.

You cannot pass unions between partitions, as parameters of events, or as parameters of methods that are started using a `START TASK` statement.

You can assign a value to only one of the members of a union data type at any one time. When you assign a value to a union data type member, remember that you can only reference the last member assigned. If you try to reference one of the other members, the results of that reference will be unpredictable.

The following table shows how you can map a union declaration between a C function and a corresponding TOOL method.

C Syntax	TOOL Syntax
union *union_name* {	**union** *union_name*
data_type member_name; . . .	[*member_name* : *data_type*;] . . .
};	**end** [**union**];

union_name is the name of the union data type within the current name scope. *member_name* is the name for each member defined for the union data type. The names of these members must be unique within the union data type. *data_type* is the data type for a member in the union.

You cannot assign an initial value for a union within the `union` syntax. Instead, you must use an assignment statement to assign the initial value.

The following example shows how you can map a C union data type declaration to a TOOL union declaration:

C Example	TOOL Example
`union PrimaryID {`	`union PrimaryID`
`   char LastName[20];`	`   LastName : array[20] of char;`
`   int SerialNumber;`	`   SerialNumber : int;`
`};`	`end union;`

After you assign the value of a union, you can refer to the member currently containing a value using this notation:

union_variable_name.member_name

union_variable_name is the name of a variable of the union data type, and *member_name* is the name of the member that has currently a value.

If you reference a union using a pointer, you can use arrow notation to dereference the value of a member of the union. To dereference the values in this union using arrow notation, use the following syntax:

pointer_to_structure_name **->** *member_name*

pointer_to_structure_name identifies a pointer to a data union, and *member_name* is the name of a member in this union.

The following example shows how you can define a union data type that can contain a value for time represented as:

- a float value specifying the number of hours in the day so far
- an array of characters that contains values like “twelve twenty-two” for 12:22
- a struct value containing two integer variables: hours and minutes

```
-- Define a union data type for time values
union myUnion
   timeHours : float;        -- Time in hours; decimal
   timeWords : pointer to char;       -- Time as text
   timeStruct : struct Inner   -- Time stored as hours and minutes
      hours : integer;
      minutes : integer;
   end;
end;
-- Declare a union variable
myVar : myUnion;
-- Assign a value to myVar
myVar.timeHours = 10.5;
...
-- Assign a different value to myVar
myVar.timeStruct.hours = 10;
myVar.timeStruct.minutes = 30;
...
-- Assign yet a different value to myVar
myVar.timeWords = strdup('ten thirty');
...
-- The following statement is not legal because the variant
-- currently assigned for myVar is myVar.timeWords
if myVar.timeStruct.hours > 12 then
   ...  -- These statements will not execute correctly
```

This example shows how you can assign values of different data types in the same area of storage, and that you should only reference the last variant assigned.

CAUTION The TOOL runtime system does not check whether a reference to a union variable variant is valid given the current value of the union variable. You are responsible for making sure that your code references the last variant assigned for the variable.

Operator Precedence and Associativity

The following table summarizes the rules of precedence and associativity for all TOOL operators. The order of precedence indicates the order in which the operators are evaluated. For example, if you have all the operators in the table in a single TOOL statement, then the -> and [] operators are evaluated first, followed by * and &, and so on.

This table includes the dereferencing operators, which are specific to C data types:

Precedence	Operators	Associativity
1	-> []	left to right
2	* (pointer dereference) & (address)	right to left
3	- (unary minus) + (unary plus)	right to left
4	* (multiplication) / %	left to right
5	+ -	left to right
6	< > = <= >= != <>	left to right
7	&	left to right
8	^	left to right
9	|	left to right
10	NOT	right to left
11	AND	left to right
12	OR	left to right

Managing Memory for C-style Arrays and Data Structures

This section explains how you can dynamically allocate storage for C-style arrays and data structures in memory that is not managed by iPlanet UDS.

iPlanet UDS provides two ways for you to declare and use C-style arrays and data structures (structs):

- Declare the C-style array or data structure as a local variable on the runtime stack.

  Use this approach when your application does not need this variable beyond the end of the current method. iPlanet UDS's memory management will automatically free the memory for this variable when the method ends. This approach is identical to the way other TOOL variables are allocated, as well as identical to the way that local or automatic variables are allocated in C.

- Declare a pointer to the C-style array or data structure, then dynamically allocate the storage using the C functions **calloc** or **malloc**.

  Use this approach when your application needs this data beyond the end of the current method. The memory allocated for this variable will remain allocated until some part of the application uses the C function **free** to explicitly free the memory. This approach is identical to the convention for allocating memory in C language programs.

  Use this approach only when necessary to retain a C-style array or data structure variable beyond the scope of the current method.

The following example shows how TOOL allocates memory for different kinds of data:

```
-- Define a struct data type
struct myStruct
   myInt : integer;
   myFloat : float;
end;
. . .
-- Define a method that declares variables
method myClass.myMethod()
begin
-- The following variables are allocated in the TOOL-managed
storage
-- area
   myNumber : integer;
   firstStruct : myStruct;
   secondPtr : pointer to myStruct;
   secondPtr = &firstStruct;
end method;
```

In this example, TOOL defines the variables myNumber, firstStruct, and secondPtr as local variables on the runtime stack each time myMethod is invoked. Note that TOOL allocates secondPtr with only enough memory for the pointer itself, not the data structure.

Dynamically Managing Memory

You should dynamically allocate memory for derived data types in your application only when the application needs to retain a C-style array or data structure variable beyond the scope of the method in which the variable is declared.

You can use the **calloc** or **malloc** functions to allocate the memory needed by a variable of a derived data type. The memory allocated for this variable will remain allocated until some part of the application uses the C function **free** to explicitly free the memory. This approach is identical to the convention for allocating memory in C language programs.

TOOL provides four C function calls for dynamically managing memory:

calloc allocates a contiguous space for a C-style array, initializes the members to zero, and returns a pointer.

free deallocates the space pointed to by a specified pointer.

malloc allocates a memory space for a value of a specified size and returns a pointer. The storage is not initialized.

strdup allocates a memory space, then copies a source string into that space and returns a pointer to char.

TOOL also provides a `sizeof` function that returns the size of a given data type.

These C functions are described in detail in the following sections.

These TOOL C functions use the corresponding C runtime library functions available for the partition where the TOOL method is running.

You can allocate and deallocate memory dynamically using these C functions either within your TOOL code or within your C application. For example, you can allocate a space using the malloc function in a TOOL method, then later deallocate the same space using the free function in a called C function.

calloc

The C function `calloc` allocates a contiguous space for a C-style array, whose members have been initialized to zero, and returns a generic pointer. The syntax for calloc is:

calloc(count=*integer***, size=***integer***) :** *pointer***;**

count=integer specifies the number of array members that you want to allocate and initialize. *size=integer* specifies the size, in bytes, that you want each member of the array to be. `calloc` returns a generic pointer to the area of initialized memory, which means that you usually need to cast the pointer to the appropriate pointer type.

```
-- Declare a pointer to an array
myPointer : pointer to array[5] of int;
-- Allocate memory for the array of 5 integers
myPointer = (pointer to array[5] of int) (calloc(5,
  sizeof(array[5] of int)));
```

Note that in this example, you need to cast the generic pointer returned by the **calloc** C function before assigning the pointer value to myPointer.

free

The C function **free** deallocates the space pointed to by a specified generic pointer. The syntax for **free** is:

free(mem=*pointer***);**

mem=*pointer* indicates a generic pointer that references the space that you want to deallocate.

```
-- Declare a pointer to an int
myPointer : pointer to int;
-- Allocate memory for the int
myPointer = (pointer to int) (malloc(sizeof(int)));
-- Do some stuff with the pointer.
...
-- When you know you don't need it anymore, free it.
free(mem = myPointer);
```

malloc

The C function **malloc** allocates a space in memory of a specified size and returns a generic pointer, but does not initialize the storage. The syntax for **malloc** is:

malloc(size=*integer***) :** *pointer***;**

size=integer is the size, in bytes, that you want the allocated space to be. **malloc** returns a generic pointer, which you usually need to cast to the appropriate pointer type.

The area of memory that is referenced by the returned pointer is not initialized, which means it might contain garbage. Do not use the value of this area of memory until you have initialized it with your own data.

```
-- Declare a pointer to an int
myPointer : pointer to int;
-- Allocate memory for the int
myPointer = (pointer to int) (malloc(sizeof(int)));
```

Note that in this example, you need to cast the generic pointer returned by calloc before you assign the pointer value to myPointer.

strdup

The C function **strdup** allocates space in memory, copies a string value into this space, and returns a pointer to char. The syntax for **strdup** is:

strdup(source=*string***) :** *pointer to char***;**

source=string specifies the string value that you want to have copied into the allocated space. **strdup** automatically allocates a space of the correct size to hold the string value.

```
-- myCharPointer is a pointer to char that references a null-
-- terminated string. TOOL stores this string as a
-- C-style array[23] of char, with 22 characters and 1 NULL.
myCharPointer : pointer to char;
myCharPointer = strdup('String of sample text.');
```

sizeof

You can determine the size of a given data type using the **sizeof** compiler function. The syntax of the sizeof function is the same as in C:

sizeof(*data_type***):** *integer***;**

data_type is the name of the data type that you want to allocate memory for.

```
-- Declare a pointer to an array
myPointer : pointer to array[5] of int;
-- Allocate memory for the array of 5 integers
myPointer = (pointer to array[5] of int) (malloc(
  sizeof(array[5] of int)));
```

Casting Pointers Returned by C Functions

The **malloc** and **calloc** C functions return a generic pointer value that points to the space allocated. To assign the generic pointer to a specific data type, you must cast the generic pointer to the appropriate type of pointer. The following example shows how you could cast a pointer:

```
-- Allocate a pointer to a structure
myPointer : pointer to myStruct;
-- Allocate memory for the structure itself
myPointer = (pointer to myStruct)(malloc(sizeof(myStruct)));
```

Managing Memory in Exception Handling

When you code the exception handlers within a method, you need to consider whether you should free allocated memory.

Generally, if a method allocates and deallocates a given area of memory, you should include in the exception handlers steps to deallocate memory:

➤ **To deallocate allocated memory in an exception handler**

1. Check whether the pointer to the C-style array or data structure is NIL.

   If the pointer is NIL, then the application has already deallocated the storage, and the exception handler does not need to do Step 2.

2. Use the `free` C function to deallocate the memory referenced by the pointer.

Managing Memory for Asynchronous Processing

To pass a C-style array or data structure as a parameter for events and methods that are started using `start task`, you must dynamically allocate the memory for the C-style array or data structure. Then, you can use a pointer to pass the address of the memory containing the C-style array or pointer.

CAUTION Make sure that your application frees the memory for these parameters only after all of the event receivers and tasks are finished using the parameter values.

For example, if you pass a pointer to a data structure as the parameter of an event, you need to make sure that the data structure exists until all possible event handlers have used the data structure.

Managing Memory Using ExternalRef Subclasses

iPlanet UDS provides a class called ExternalRef that allows the iPlanet UDS system to automatically manage when external resources are deallocated, based on when TOOL code no longer references the external resources.

The ExternalRef class is part of the Framework library. This class is an abstract class that you can use in your TOOL code to reference external resources, such as files and data structures. You must subclass the ExternalRef class to define the kind of resources being managed. An instance of a subclass of ExternalRef represents an external resource.

When a TOOL object wants to reference an external resource represented by an instance of an ExternalRef subclass, the TOOL object is bound to that ExternalRef subclass instance. When all the bound TOOL objects have been released by iPlanet UDS memory management because they are no longer required, the iPlanet UDS system invokes the `Release` method of the instance of the ExternalRef subclass. The `Release` method releases the external resource. After the `Release` method completes, the iPlanet UDS system releases the instance of the ExternalRef subclass.

The `XRefTime` example demonstrates how you could use an ExternalRef subclass to reference external resources.

For complete information about defining and using ExternalRef subclasses, see the iPlanet UDS online Help.

Mapping C Function Parameters in TOOL Methods

This section describes how to map the ways that TOOL passes parameters to your external programs.

When you write a project definition, such as a C project, in a file, you need to map the C function parameters to the parameters of the corresponding TOOL method. For this discussion, we will call the TOOL method that maps to the C function the *mapping method*.

You can use three TOOL options on the mapping method parameters: `input`, `output`, and `input output`. These options specify how data is passed between TOOL and the C function. Because C passes and returns all parameters as values, you need to decide which TOOL options to specify based on what values you actually want to pass, and in which direction. The `input`, `output`, and `input output` options are described in "Specifying TOOL Parameter Options" on page 246.

You must understand exactly what the C function intends to pass and why. You must also understand whether the *calling method*, the method that calls the C function using the mapping method, will later use this parameter.

The following table shows you how you can map C function parameters to parameters in TOOL mapping methods. When a C function header can be mapped in more than one way, then you might need to consider how you would call this function using C to determine which mapping to use. These mappings are explained later in this section.

C Function Parameter Type	C Function Header Example	How C Function is Called	TOOL Mapping Method	See:
Simple data type	CFn(int MyParm)	CFn(IntValue)	CFn(input MyParm:int);	page 241
Pointer to a simple data type	CFn(int *MyParm)	CFn(PtrInt) CFn(PtrInt) CFn(&IntValue)	CFn(input MyParm: pointer to int); CFn(output MyParm: int); CFn(input output MyParm: int);	page 241
Data structure	CFn(struct sType MyParm); CFn(struct sType *MyParm); CFn(struct sType *MyParm); CFn(struct sType *MyParm);	CFn(MyStruct) CFn(PtrStruct) CFn(PtrStruct) CFn(&MyStruct)	CFn(input MyParm: sType); CFn(input MyParm: pointer to sType); CFn(output MyParm : sType); CFn(input output MyParm: sType);	page 243
Array	CFn(char *PtrArray); CFn(char MyArray[10]); CFn(int MyArray[]); CFn(char **PtrArray);	CFn(PtrArray) CFn(MyArray) CFn(MyArray) CFn(*PtrArray)	CFn(input PtrArray : pointer to char); CFn(input MyArray[10] of char); CFn(input MyArray[] of int); CFn(output PtrArray: pointer to char);	page 244

Mapping Simple C Data Type Parameters

In general, if the C parameter is a simple data type, like int or char, then the TOOL parameter is an input parameter of the corresponding TOOL type.

The following example shows how a C function with a parameter of a simple data type would be defined as a TOOL method:

C Function	TOOL Method
`void op1(int firstParm);`	`op1(input firstParm : int);`

Mapping Pointer Parameters

C functions use pointers as parameters to:

- pass data by reference
- pass addresses that can be changed
- pass output values back to the calling application

You need to know how the C function uses the parameters to decide whether the TOOL parameter that maps to the pointer should be a `input` or `output` pointer parameter or an `input output` parameter.

You can frequently determine which option to use based on the value you would pass to this C function from another C function.

The examples in this section work with the following function header:

```
void MyFunction(int *MyParm);
```

Passing an Input Value with the Pointer

In C, if you would normally pass an input pointer value when you invoke the C function, then for the TOOL mapping method, you should use the `input` option and define the mapping method parameter as a pointer.

In this case the C function pointer maps to an `input` pointer parameter in the mapping method, as shown:

C Function	TOOL Equivalent
`void MyFunction(int *MyParm);`	`MyFunction(input MyParm : pointer to int);`

In this example, MyFunction either uses a copy of the value referenced by the pointer, or dereferences the pointer and changes the value referenced by the pointer.

Getting an Output Value using the Pointer

In C, if you would normally pass the address of an int variable (&MyParm) to the C function to retrieve a value produced by the C function, then you should use the `output` option. You do not need to specify the pointer in this case, because the `output` option makes TOOL implicitly pass the address of the variable.

In this case the C function pointer parameter maps to an `output` parameter in the mapping method, as shown:

C Function	TOOL Equivalent
`void MyFunction(int *MyParm);`	`MyFunction(output MyParm : int);`

In this example, MyFunction generates a result and passes the address of the result back to the calling application using the pointer parameter.

Passing an Input Value That Will Change

In C, if you would normally pass an address (&*variable*) as an input parameter when you invoke the C function, then you should specify the `input output` option with the name of the variable. You do not need to specify the pointer in this case, because the `input output` option makes TOOL implicitly pass the address of the variable.

In this case the C function pointer would map to an `input output` parameter with one less level of indirection in the mapping method, as shown:

C Function	TOOL Equivalent
`void MyFunction(int *MyParm);`	`MyFunction(input output MyParm : int);`

In this example, MyFunction accepts an address as an input value, changes the address or value at the address, then passes back the final value of the address to the calling application.

Mapping Data Structure Parameters

The following examples show how you can map C function parameters involving data structures to TOOL method parameters. These examples omit the return value.

How the C Function Uses the Struct Parameter	C Function Header	TOOL Mapping Method
Needs a copy of the data structure	CFn(struct stype MyParm);	CFn (input MyParm : sType);
Dereferences and changes the data structure	CFn(struct stype *MyParm);	CFn (input MyParm : pointer to sType);
Produces an output data structure	CFn(struct sType *MyParm);	CFn (output MyParm : sType);
Changes and passes back the data structure	CFn(struct sType *MyParm);	CFn (input output MyParm : sType);

Mapping C-Style Array Parameters

TOOL methods pass C-style arrays by reference. The TOOL syntax for passing a C-style array is functionally equivalent to passing a pointer to the first element of the array. However, you probably want to map the syntax of your TOOL mapping method as closely as possible to the C function header.

You can only pass C-style arrays as input parameters. If you need to retrieve an array from a C function (an output or input output parameter), you must pass a pointer to an array.

NOTE If you are retrieving an array of characters from a C function, you can pass a string; however, in this case the value is stored in memory managed by iPlanet UDS, so you cannot rely on the address of the string value remaining constant after the C function returns. It is better to pass a pointer to char instead of string, because the memory address for the array of char remains constant until the memory is deallocated.

The following examples shows the mapping of the C function to their closest TOOL equivalents:

C Function	TOOL Equivalent
`CFunction(char *PtrToArray);`	`CFunction(input PtrToArray : pointer to char);`
`CFunction(char MyArray[10]);`	`CFunction(input MyArray : array[10] of char);`
`CFn(char **PtrTCharArray);`	`CFn(output PtrCharArray:pointer to char);`
`CFn(char **PtrTCharArray);`	`CFn(output PtrToCharArray : string);`

Mapping Return Values

When you map a C function header to a TOOL mapping method, you must also map the return value of the C function as a return value for the mapping method. The following table shows how to map the return values of the C function to the TOOL mapping method.

C Function Header Syntax	TOOL Mapping Method Syntax
return_type function_name ([*data_type parameter,*]);	*function_name* ([*parameter* **:** *data_type;*] . . .) [**:** *return_type*] **;**

return_type is the data type of the return value.

The following example shows how you would map the return value of a C function header to the return value for a TOOL mapping method:

C Function Header Example	TOOL Mapping Method Example

`struct tm *localtime(`
`  const time_t *tp)`
	`localtime(`
	`  input a: pointer to time_t`
	`) : pointer to tm`

In this example, the return value in both cases is a pointer to a tm struct.

In general, you follow the same rules for mapping C data types to TOOL data types for the return values as for function parameters. For more information, see "Mapping Simple C Data Types to TOOL Data Types" on page 201 or "Mapping Derived C Data Types to TOOL Data Types" on page 203.

Specifying TOOL Parameter Options

You can use three TOOL options on the mapping method parameters to set the TOOL mechanism for passing parameters: `input`, `output`, and `input output`. These options specify how data is passed between TOOL and the C function. Because C passes and returns all parameters as values, you need to decide which TOOL options to specify based on what values you actually want to pass, and in which direction. The following sections describe how these mechanisms work with C functions.

Input Mechanism

By default, parameters in TOOL methods are input parameters. When a parameter in the mapping method has the `input` option, TOOL passes a copy of the value to the C function. Generally, changing the value of the parameter in the C function does not affect the value in the calling method. However, if you pass a pointer to a derived data type as an input parameter, you can dereference the pointer and change values in the derived data type. The calling method can later reference the same pointer to see the changed values.

The following examples show how you can use input parameters and how they work. In these examples, v2 is an object of the class that contains the methods shown.

```
-- In C, the C function header for inputInt() is:
-- char *inputInt(int a1)

-- Declare the mapping method:
inputInt(a : int) : string;

-- Call this method with TOOL code:
intVal : int = 20;
v2.inputInt(intVal);
-- Whatever happens to intVal in inputInt is not
-- reflected here in the calling method.
```

```
-- In C, the C function header for inputPointer() is:
-- char *inputPointer(void *a1)

-- Declare the mapping method:
inputPointer(a : pointer) : string;

-- Call this method with TOOL code:
intVal : int = 20;
ourPtr : pointer to int;
ourPtr = &intVal;
-- Now call the inputPointer method with this pointer.
v2.inputPointer(ourPtr);
-- We can't assume that intVal is still 20 after this call.
```

You can use the iPlanet UDS example program AllCType as a reference for how to define mapping methods for parameters of all data types and levels of indirection.

Output Mechanism

When a parameter in the mapping method has the `output` option, TOOL retrieves a value from the C function when the C function completes. The calling method does not provide an input value for this parameter when it invokes the C function.

In C, a C function can only pass back the value of a parameter when the parameter is a pointer. Therefore, you can only pass back values by reference, using the value of this pointer.

However, in TOOL, the `output` option tells TOOL to expect the address of the parameter. When the C function passes back the address of the parameter, TOOL automatically dereferences this address and passes the parameter itself to the calling method.

The `output` option automatically passes an address value to the C function. Therefore, you must remove one level of indirection when you map a pointer parameter in the C function to a TOOL mapping method.

The following examples show how you can use output parameters and how they work. In these examples, v2 is an object of the class that contains the methods shown.

```
-- In C, the C function header for outputInt() is:
-- char *outputInt(int *a1)
-- because output parameters are passed by reference.

-- Declare the mapping method:
outputInt(OUTPUT a : int) : string;

-- Call this method with TOOL code:
intVal : int;
v2.outputInt(intVal);
-- intVal is assigned a value in outputInt().
```

```
-- In C, the C function header for outputPointer() is:
-- char *outputPointer(int **a1)
-- because all output parameters are passed by reference,
-- including pointers.

-- Declare the mapping method:
outputPointer(OUTPUT a : pointer to int) : string;

-- Call this method with TOOL code:
ourPtr : pointer to int;  -- the value of this pointer is NIL
v2.outputPointer(ourPtr);
-- ourPtr is assigned a value in outputPointer().
```

You can use the iPlanet UDS example program AllCType as a reference for how to define mapping methods for parameters of all data types and levels of indirection.

Input Output Mechanism

When a parameter of the mapping method is specified using the `input output` option, the calling method passes an input value to the C function. The calling method expects the C function to change the value of this parameter. When the C function returns control to the calling method, TOOL assigns the value of the parameter in the C function to the value passed as the parameter in the calling method.

In TOOL, the `input output` option tells TOOL to pass the address of some copy of the specified parameter to the C function. When the C function passes back the address of the final value of the parameter, TOOL automatically dereferences this address and assigns the final value to the parameter in the calling method.

The `input output` option automatically passes an address value to the C function. Therefore, you can remove one level of indirection when you map a pointer parameter in the C function to a TOOL mapping method.

The following examples show how you can use input output parameters and how they work. In these examples, v2 is an object of the class that contains the methods shown.

```
-- The C function outputInt() is defined like this:
-- char *ioInt(int *a1)
-- because input output parameters are passed by reference.

-- Declare the mapping method like this:
ioInt(INPUT OUTPUT a : int) : string;
-- Note that you remove one level of indirection for the parameter

-- Call this method with TOOL code like this:
intVal : int;
v2.ioInt(intVal);
-- intVal now has whatever value it was assigned in ioInt().
```

```
-- The C function ioPointer() is defined like this:
-- char *ioPointer(int **a1)
-- because all output parameters are passed by reference,
-- including pointers!

-- Declare the mapping method like this:
ioPointer(INPUT OUTPUT a : pointer to int) : string;
-- Note that you remove one level of indirection for the parameter

-- Call this method with TOOL code like this:
ourPtr : pointer to int;
v2.ioPointer(ourPtr);
-- ourPtr has whatever value it was assigned in ioPointer().
```

You can use the iPlanet UDS example program AllCType as a reference for how to define mapping methods for parameters of all data types and levels of indirection.

Part 4

Writing C++ Client Applications

Part 4 of *Integrating with External Systems* provides complete information about integrating with C++.

Part 4 contains the following chapters:

Chapter 14

Accessing iPlanet UDS Using C++

This chapter explains how you can generate a C++ API that lets you access your iPlanet UDS application using C++ calls.

This chapter includes the following topics:

- designing an application to be accessed by C++
- generating a C++ API for an iPlanet UDS application
- writing a C++ client application that accesses an iPlanet UDS application
- writing a C++ client application that accesses the iPlanet UDS runtime system

About Accessing iPlanet UDS Using C++

This chapter explains how to produce and use a C++ API (application program interface) that provides access to an iPlanet UDS application and how to access the runtime library using the C++ libraries provided as part of the iPlanet UDS product.

iPlanet UDS lets you develop C++ client applications that interact with both iPlanet UDS applications and the iPlanet UDS runtime system itself. iPlanet UDS provides a C++ API to all the classes, attributes, and methods defined by the iPlanet UDS libraries, except for the Display library. You can easily generate a C++ API for any iPlanet UDS application that has a client partition. iPlanet UDS cannot generate C++ APIs for server applications (applications with no client partitions) or directly for service objects.

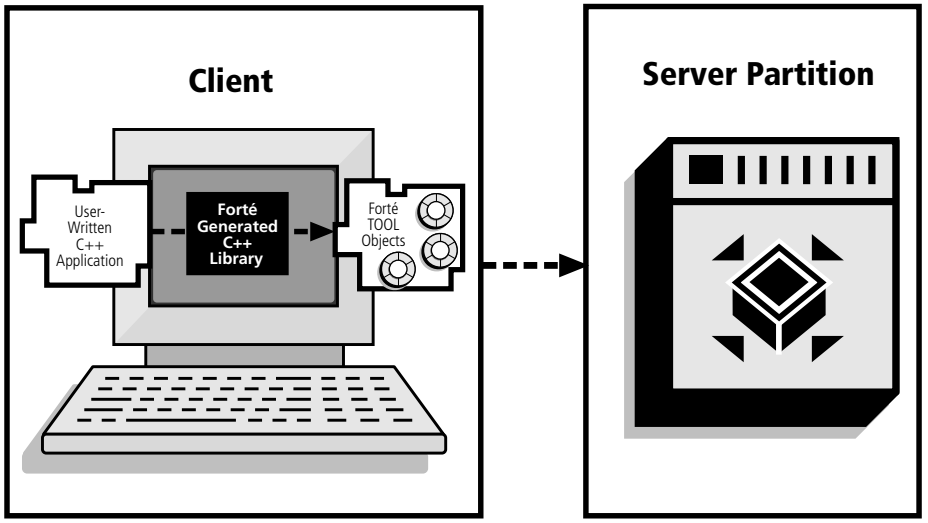
C++ applications cannot access objects of the Display library classes or their subclasses, such as windows or widgets.

NOTE You cannot access C++ client applications or functions from within iPlanet UDS.

Using the C++ APIs, you can write C++ code that interacts with the iPlanet UDS classes to implement servers, database access applications, and so forth. We strongly recommend that you take advantage of the features provided by iPlanet UDS to implement these kinds of applications, then write C++ client applications that access these iPlanet UDS services.

The C++ client application starts the iPlanet UDS client partition and controls the flow of the application. This C++ client application then instantiates classes and invokes methods in the iPlanet UDS application and the iPlanet UDS runtime system.

Figure 14-1 Using a C++ API



To generate a C++ API for a client application, you set a property on the client partition and compile the generated C++ code.

For detailed instructions about how to generate a C++ API for an application, see "Generating a C++ API for an iPlanet UDS Application" on page 259.

Use handle classes The C++ API header file that is generated contains definitions of *handle classes* that are implemented in the C++ API. The handle classes represent the classes in the main project of the application and the classes in the supplier plans that are used by the application.

As a C++ programmer, you create or reference objects using these handle classes. You can then interact with the iPlanet UDS TOOL objects using these handle classes.

Functions provide handles to service objects Any service objects that are defined in the main project of the iPlanet UDS client application or that are accessible in the supplier plans are accessible using global functions defined in the C++ API header file.

Access attributes using methods You cannot directly access the attributes of iPlanet UDS objects using a generated C++ API. Instead, iPlanet UDS generates get and set member functions that you can use to retrieve values and set attributes for methods. You cannot access virtual attributes using the C++ API.

NOTE C++ client applications cannot directly register for events that are posted by iPlanet UDS or user applications. C++ clients also cannot directly post events. However, you can define TOOL methods that post events and register for events, then generate C++ APIs for these methods. This approach is described in "Events" on page 296.

Terminology Used in Part 3

This section contains terms that this manual uses to describe how you can access iPlanet UDS services using C++.

C++ API An application program interface that lets a C++ client application access services and use runtime functions provided by an iPlanet UDS application.

C++ client application A C++ application that uses the C++ API for an iPlanet UDS client application.

Handle class A class that provides external applications a safe API to a regular iPlanet UDS class.

Handle object An instance of the handle class that represents an object in the iPlanet UDS client application.

Task handle A reference to a task running in the iPlanet UDS client application.

Designing an Application to be Accessed by C++

Any iPlanet UDS application that you want to access using C++ must have a client partition. Your C++ client application interacts with the iPlanet UDS client partition, and iPlanet UDS manages communication with other iPlanet UDS services. iPlanet UDS uses the start class and method for the iPlanet UDS client partition to determine what handle classes and global functions to generate.

Although you can choose to generate a C++ API directly from an existing client partition in your application, in most cases, it is useful to define a client partition specifically for generating a C++ API.

Restrictions when Generating and Using a C++ API

This section describes some restrictions you should consider before producing a C++ API.

C++ API Uses Case Defined in TOOL

Although TOOL is not case sensitive, C++ is, and the C++ API that is generated based on your iPlanet UDS application keeps the same case that you used in your TOOL code.

No Virtual Attributes

iPlanet UDS does not include virtual attributes or their methods in the generated C++ handle classes.

Cannot Use Subclasses of Display Library Classes

You can generate a C++ API for a partition that uses classes and subclasses from the Display library; however, you can not access classes that are classes and subclasses of the Display library.

No C++ API for Events

iPlanet UDS does not generate C++ APIs for registering for iPlanet UDS events or for posting iPlanet UDS events. If you want a C++ client application to be able to respond to iPlanet UDS events, you should define a TOOL method that contains an event loop that registers for the event. This approach is described in "Events" on page 296.

Supplier Libraries Must Be Compiled and Have Handle Classes

If the C++ client application uses supplier TOOL libraries, these TOOL libraries must have their handle classes available and be compiled libraries. If the supplier libraries are not compiled and do not have these handle classes available, applications that use the C++ API will have unpredictable runtime behavior.

If the supplier TOOL libraries have not been compiled and do not have their handle classes available, you need to make a new compiled distribution for these libraries while running Fscript or the Partition Workshop with the cg:13:1 configuration flag set.

Defining a Client Partition for the C++ API

In many cases, the existing client partition for your client application will not produce an appropriate set of C++ class definitions. The client partition might produce too many classes in its API, or you might not want C++ client applications to be able to interact with certain classes or service objects using certain methods. You can define a client partition specifically for C++ clients to make the generated C++ classes more usable for C++ programmers and limit access to certain services in your application.

To generate a C++ API that accesses a server application (an application containing only server partitions), define a client partition for the application. You can also create a custom iPlanet UDS client for an existing client application, so that you can customize the C++ API that gets generated.

For example, suppose the existing client partition for an application contains several windows and only accesses a few of the services that you want a C++ client application to access. In this case, you can create a new main project that defines the new client partition. In this project, you can define the plans for several unrelated services as suppliers to this main project and simply invoke the Init methods on the service objects from the various supplier plans.

The example used in this section generates classes and global functions for the services provided by the BankServices project.

➤ **To define a new client partition**

1. Create a new project.

   In the example, create a project called CppBank, which does not include the Display or GenericDBMS libraries.

2. Include the supplier projects that define the service objects that you want to make available to C++ client applications.

   In the example, add a supplier project called BankServices.

3. Define a nonwindow class.

   In the example, define a class called `CppAPI`.

4. Define a method.

   In this method, instantiate each class that you want to include from the supplier classes and invoke a method on each service object that you want to access from C++.

   In the example, the BankingAccess has a method that references the BankServer service object to ensure that handle classes are generated for the classes and methods used by this service object.

```
super.Init();
s1 : BankService  = BankServer;
```

Project: CppBanking • **Class:** BankingAccess • **Method:** Init

5. Define this method as the start method for the project.

6. Configure a client application using this project as the main project.

Generating a C++ API for an iPlanet UDS Application

You need to generate a C++ API for each platform on which you intend to have a C++ client application interact with the iPlanet UDS application.

The steps you need to perform depend on whether you use the auto-compile feature. You generate the C++ API for a client partition at the same time iPlanet UDS would generate code for a compiled client partition. The auto-compile feature generates and compiles the files for the C++ API when you make a distribution. If you do not use the auto-compile feature, the `fcompile` command generates and compiles the files for the C++ API.

When you make a distribution using the auto-compile and auto-install features, the files that are generated and installed include the usual image repositories or executables for the client partition, as well as the files that enable the C++ API.

These files are installed in the same `FORTE_ROOT/userapp` subdirectory in which the client partition files are installed. For example, if the application is called CppBanking with a release of cl0, and then the C++ API files and the client partition files are in `FORTE_ROOT/userapp/cppbanki/cl0/`.

The files that are generated and installed for the C++ API are described in "Files Generated as Part of a C++ API" on page 285.

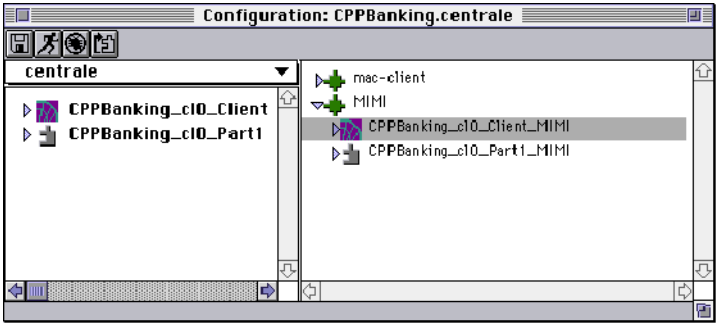
The following sections describe how to generate a C++ API using the iPlanet UDS workshops or Fscript commands.

Partition the Application

Assign the client partition wherever you want to generate a C++ API.

➤ **To partition the application using the iPlanet UDS Workshops**

1. Click the File > Partition command to partition the application and open the Partition Workshop.
2. In the Partition Workshop, assign the client partition wherever you want to generate a C++ API.



➤ **To partition the application using Fscript**

1. Enter the following series of Fscript commands:

```
fscript> FindPlan plan_name # Name of main project.
fscript> FindActEnv
fscript> Partition 3  # Replace any current configuration.
fscript> AssignAppComp node_name  component_name
```

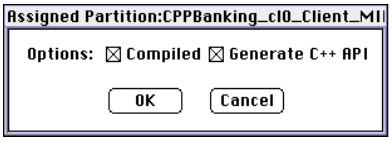
For more information about these commands, see the *Fscript Reference Guide*.

Set the Compiled and Client Partition Options

On the application's client partition, set the properties that indicate that a C++ API is to be generated and compiled. You need to set these property for each assigned client partition where you want a C++ API generated and compiled.

➤ **To set the partition options using the iPlanet UDS Workshops**

1. On each client node for which you want a C++ API generated, open the properties dialog.
2. Toggle the Compiled and Generate C++ API toggles on, then click OK to close the dialog.



➤ **To set the partition options using Fscript**

1. Enter the following series of Fscript commands:

```
fscript>SetAppCompCompiled node_name 1 component_name 1
fscript> Partition
```

For information about Fscript and the `SetAppCompCompiled` command, see the *Fscript Reference Guide*.

Make the Distribution

The steps you need to perform depend on whether or not you use the auto-compile feature.

NOTE If you intend to deploy both a compiled iPlanet UDS client partition and a C++ API based on the same partition on the same platform, you need to make two distributions, then install, as described in Technote 11129.

Using the Auto-compile and Auto-install Feature

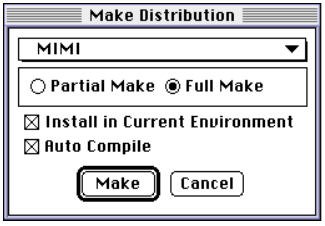
If you use the auto-compile and auto-install features, this step also generates the files for the C++ API, as well as the C++ code for any compiled partitions. iPlanet UDS next compiles the C++ API and any compiled partitions, then installs the application in the appropriate place in the `FORTE_ROOT/userapp` directory.

If you use the auto-compile and auto-install features, you do not need to perform steps described under "Compile and Install (If Auto-compile and Auto-install Are Not Used)" on page 263.

To use the auto-compile feature, your system manager must have installed the CodeGenerationSvc and AutoCompileSvc applications appropriately in your environment. For information about installing these applications, see the *iPlanet UDS System Management Guide*.

➤ **To make a distribution using the iPlanet UDS Workshops**

1. Click the File > Make Distribution command. When the Make Distribution dialog appears, select the location for the application distribution.



2. To use the auto-compile and auto-install features, toggle the Auto-Compile and Install in Current Environment toggles to on.
3. Click OK.

iPlanet UDS makes the application distribution.

➤ **To make a distribution using Fscript**

1. To use the auto-compile and auto-install features, enter Fscript commands as shown

```
fscript> MakeAppDistrib 1 node_name 1 1
```

2. To make a distribution without the auto-compile and auto-install features, enter Fscript commands as shown:

```
fscript> MakeAppDistrib 1 node_name 0 0
```

Compile and Install (If Auto-compile and Auto-install Are Not Used)

You need to perform these steps only when you do not use the auto-compile and auto-install features when you make the distribution.

You perform these steps outside Fscript and the iPlanet UDS workshops, and the steps are the same regardless of which method you used to make the distribution.

➤ **To compile and install a C++ API**

1. For each platform on which you want the C++ API available, copy the .pgf file to a node on which you have iPlanet UDS and the appropriate compiler installed.

2. Use the `fcompile` command, as described in "Using the fcompile Command to Generate the C++ API" on page 264, to generate the files for the C++ API, as well as the C++ code for any compiled partitions. iPlanet UDS then compiles the C++ API and any compiled partitions.

3. Copy the generated and compiled files from the node where you used the `fcompile` command back to the appropriate subdirectory under `FORTE_ROOT/appdist`:

   `FORTE_ROOT/appdist/`*`environmentID`*`/`*`distributionID`*`/cl#/`*`platformID`*`/`

   For example if the CppBanking example is compiled for the Windows NT platform in an environment called CentralEnv, then the directory is `FORTE_ROOT/appdist/centrale/CppBanki/cl0/pc_nt`.

4. Using the Environment Console or the Escript utility, load the distribution and install the application.

   With the Environment Console, use the File > Load Distribution command, then in the Application View, select application and use the Utility > Install command.

   Using the Escript utility, enter a sequence of commands like the following:

```
escript> ShowAgent
escript> ListDistribs
escript> LoadDistrib application_name  compatibility_level
escript> Install
```

Using the fcompile Command to Generate the C++ API

The `fcompile` command has the following syntax when used for compiling C++ libraries and compiled partitions:

Portable syntax (all platforms)

fcompile [**-c** *component_generation_file*] [**-d** *target_directory*]

[-**cflags** *compiler_flags*] [**-lflags** *linking_flags*]
[**-fm =** *memory_flags*] [**-fl =** *logger_flags*] [**-cleanup**]

OpenVMS syntax

VFORTE FCOMPILE
[**/COMPONENT =** *component_generation_file*]
[**/DIRECTORY =** *target_directory*]
[**/COMPILER =** *compiler_flags*]
[**/LINKING =** *linking_flags*]
[**/MEMORY =** *memory_flags*]
[**/LOGGER =** *logger_flags*]
[**/CLEANUP**]

The following table describes the command line flags for the `fcompile` command:

Flag	Description
-c *component_generation_file* **/COMPONENT =** *component_generation_file*	Specifies the file that iPlanet UDS compiles. This value includes the path where the file resides if the file is not in the current directory. By default, iPlanet UDS compiles all files in the current directory.
-d *target_directory* **/DIRECTORY =** *target_directory*	Specifies where the compiled directories will be placed. By default, `fcompile` compiles files in the current directory, and places the compiled files in the current directory. *target_directory* is a directory specification in local syntax. If the **-c (/COMPONENT)** flag is also specified, the **-d** flag specifies where the compiled component files will be placed. Otherwise, the directory specified by the **-d (/DIRECTORY)** flag specifies both the directory containing the files to be compiled and the directory where the compiled files will be placed.
-cflags *compiler_flags* **/COMPILER =** *compiler_flags*	Specifies any C++ compiler options.
-lflags *linking_flags* */LINKING = linking_flags*	Specifies any linking flags.
-fm *memory_flags* **/MEMORY =** *memory_flags*	Specifies the space to use for the memory manager. See *A Guide to the iPlanet UDS Workshops* for information.
-fl *logger_flags* **/LOGGER =** *logger_flags*	Specifies the logger flags to use for the command. See *A Guide to the iPlanet UDS Workshops* for information.
-cleanup **/CLEANUP**	Deletes all the files except for the newly compiled shared libraries.

Writing a C++ Client Application That Accesses an iPlanet UDS Application

This section summarizes the steps for writing, compiling, linking, and editing a C++ client application that accesses an iPlanet UDS application. These steps will be slightly different for each platform.

This section also outlines the capabilities and restrictions that you need to be aware of to write a C++ client application that accesses an iPlanet UDS application.

Handle classes Handle classes and their member functions allow you to write C++ client applications that interact with iPlanet UDS objects and runtime processes without having to worry about the internals of how iPlanet UDS manages objects. For example, the handle classes ensure that you can write C++ code that interacts with iPlanet UDS objects without needing to know:

- where a distributed object is located
- how and when memory is allocated and deallocated when you create and destroy iPlanet UDS objects
- where objects are stored when iPlanet UDS reclaims memory using its memory reclamation (garbage collection) facilities

NOTE Although TOOL is not case sensitive, C++ is, and the C++ API that is generated based on your iPlanet UDS application keeps the same case that you used in your TOOL code.

The first action your C++ code must perform is to define an instance of the handle class for a task handle, qqhTaskHandle, which represents the main task accessible in the client partition. Next, you start iPlanet UDS using the global function ForteStartup, which starts an iPlanet UDS task associated with this C++ process, and initializes the client partition. This step is described in detail in "Start iPlanet UDS Interaction" on page 274.

Using handle classes and methods When you write a C++ client application that uses the C++ APIs for the iPlanet UDS applications or runtime system, you can use the handle classes defined in the header file to create and destroy instances of iPlanet UDS objects. After you have created or assigned an iPlanet UDS object to a new variable of a given handle classes, you can call member functions on this object, just as you would call the methods if you were using TOOL. This step is described in detail in "Using Handle Classes and Methods" on page 277.

Interacting with service objects The C++ API header file contains global functions that return handles to service objects. These global functions have the same names as the service objects. This step is described in detail in "Interacting with Service Objects" on page 276.

In your C++ client application, you can create multiple threads and associate each thread with an iPlanet UDS task so that each thread can interact with iPlanet UDS.

Interacting with the iPlanet UDS runtime system iPlanet UDS provides handle classes for all classes that are used by the iPlanet UDS client partition, including classes in supplier plans to the main project. This step is described in detail in "Interacting with the iPlanet UDS Runtime System" on page 279.

iPlanet UDS provides handle classes and member functions in the `FORTE_ROOT/install/inc/handles/` directory for all the classes in the Framework library, including classes that enable you to interact with the iPlanet UDS runtime system.

Shutting down the iPlanet UDS client partition When your C++ client application has finished using iPlanet UDS, you can use the ForteShutdown global function to shut down the client partition. The ForteShutdown global function does not automatically shut down running server partitions. This step is described in detail in "Shutting Down the iPlanet UDS Client Partition" on page 279.

Understanding the C++ API

iPlanet UDS generates the following files, which are all critical for understanding the C++ API:

***client_component_id*.txt** A road map explaining how to find particular handle class definitions, global functions for service objects, and so forth.

***client_component_id*.h** The main header file for the C++ API. This file includes all other needed header files except for those containing handle class definitions for the C++ API (.cdf files). This file also contains the global functions that access service objects.

c1.cdf, c2.cdf, and so forth Files containing class definitions for the handle classes that are part of the C++ API.

The following sections explain how to use these files to understand the iPlanet UDS service objects, classes, attributes, and methods are available through the C++ API.

Getting an Overview: *client_component_id*.txt

The following example shows the *client_component_id*.txt file that would be generated for the CPPBanking C++ API:

```
Readme for C++ API for partition: cppban0

This file describes the files and classes that
have been generated for using this C++ API

Project: BankServices

TOOL Class      File Name Handle Class       Forte Internal C++ Class
-----------------------------------------------------------------
AccountNotFound c5.cdf    qqhAccountNotFound AccountNotFound_c5
BankAccount     c4.cdf    qqhBankAccount     BankAccount_c4
BankService     c3.cdf    qqhBankService     BankService_c3

Project: CPPBanking

TOOL Class      File Name Handle Class       Forte Internal C++ Class
-----------------------------------------------------------------
BankingAccess     c6.cdf      qqhBankingAccess     BankingAccess_c6

Service Objects

Name          Class              Function to retrieve service object
-----------------------------------------------------------------
BankServer    qqhBankService     BankServer();
```

You can use the *client_component_id*.txt file to understand what handle classes (and corresponding TOOL classes) can be accessed by C++ clients. The names of handle classes start with "qqh" and end with the name of the corresponding TOOL class. For example, the handle class called qqhBankAccount corresponds to a TOOL BankAccount class.

NOTE Some of the c#.cdf files are used only by iPlanet UDS, and are therefore not listed in the *client_component_id*.txt file for this C++ API. Do not use classes defined in files not listed in the *client_component_id*.txt file.

Locating Global Functions: *client_component_id*.h

This file contains global functions that return handles to iPlanet UDS service objects that the C++ API can access.

For example, the following example shows the global function defined for the BankServices.BankServer service object:

```
qqEXPORTFUNCTION(qqhBankService) BankServer();
```

The *client_component_id*`.h` file is the main header file for the C++ API. This file includes all the header files that this API requires, except for the `c#.cdf` files that are described in the *client_component_id*`.txt` file. You need to include each `c#.cdf` file separately, as described in "Locating Class Definitions: c#.cdf" on page 269.

qqEXPORTFUNCTION indicates that this function is available to other applications, such as a C++ client application. Before you use this function in your C++ code, you need to declare the function using the extern keyword, as shown in the following example:

```
extern qqhBankService BankServer();
```

You need to include this file (#include) in your C++ client application, as shown in the following example:

```
#include CppBank.h
```

Locating Class Definitions: c#.cdf

The c#.cdf files contain the class definitions for the handle classes and C++ classes that represent the TOOL classes. One .cdf file is generated for each TOOL class. (.cdf stands for class definition file.) Check the *client_component_id*`.txt` file to determine which .cdf contains the class definition for the handle class you want.

The class definition for the handle class is usually in the second half of the file. The class statement for the handle class contains "class qqEXPORTCLASS qqh*classname*".

You need to include each c#.cdf file that contains the class definition for a handle class you are using in your C++ client application, as shown in the following example:

```
#include c4.cdf
```

The following example shows the class definition for a handle class in a `c#.cdf` file:

```
class qqEXPORTCLASS qqhBankService : public qqhObject
{
public:
  qqhBankService();
  qqhBankService(const qqhBankService& other);
  qqhBankService(BankService_c5*);
  ~qqhBankService();
  qqhBankService& operator=(const qqhBankService& other);
  operator BankService_c5*() const;
  void New(const qqhTaskHandle& task);

  // Attribute Get/Set pairs
  qqhArray AcctList(const qqhTaskHandle& task);
  void AcctList(const qqhTaskHandle& task, const qqhArray&
value);

  // Methods

  void Init(const qqhTaskHandle& task);
  double UpdateAcct(
    const qqhTaskHandle& task, qqos_i4 acctNumber,
    double transactionAmt);
  qqhBankAccount GetAcctData(const qqhTaskHandle& task,
    qqos_i4 AcctNumber);
  qqhArray GetAcctNumList(const qqhTaskHandle& task);
};
```

qqEXPORTCLASS indicates that this class is available to other applications, such as a C++ client application.

NOTE Some of the c#.cdf files are used only by iPlanet UDS, and are therefore not listed in the *client_component_id*.txt file for this C++ API. Do not use classes defined in files not listed in the *client_component_id*.txt file.

Setting up Your System and Compiler to Use the C++ API

There are a few steps you should take to ensure that all the correct iPlanet UDS and C++ API files can be located by the compiler and linker when you compile and build your C++ client application:

➤ **To set up your system and compiler**

1. Edit your library search path to specify the directory containing the shared library file for the C++ API.

   For example, the shared library file for the C++ API for the CPPBanking application is installed in `FORTE_ROOT/userapp/cppbanki/cl0`, so you should include this directory in your library search path.

2. Set your include path to specify the following directories, which contain header files for iPlanet UDS runtime and library classes:

   - `FORTE_ROOT/install/inc/cmn`
   - `FORTE_ROOT/install/inc/ds`
   - `FORTE_ROOT/install/inc/handles`
   - `FORTE_ROOT/install/inc/os`
   - the directory containing the .h and .cdf files, which is the `FORTE_ROOT/userapp` subdirectory containing the files for your client partition

   For example, the .h and .cdf files for the C++ API for the CPPBanking application are installed in `FORTE_ROOT/userapp/cppbanki/cl0`, so you must include this directory in your include path.

3. Using an environment variable or the make file, specify the libraries needed for linking.

On Windows NT and Windows 95, the libpath contains a list of .lib files, and you need to specify the following:

- `FORTE_ROOT\install\lib\qqhd.lib`
- `FORTE_ROOT\install\lib\qqsm.lib`
- `FORTE_ROOT\install\lib\qqfo.lib`
- `FORTE_ROOT\install\lib\qqdo.lib`
- `FORTE_ROOT\install\lib\qqsh.lib`
- `FORTE_ROOT\install\lib\qqcm.lib`
- `FORTE_ROOT\install\lib\qqkn.lib`
- the directory containing the .lib file, which is installed in the `FORTE_ROOT\userapp` subdirectory containing the files for your client partition

For example, the .lib file for the C++ API for the CPPBanking application is `FORTE_ROOT\userapp\cppbanki\cl0\cppban0.lib`, so you must include this file in your libpath.

On UNIX platforms, the path that specifies libraries for linking contains a list of shared library files, and you need to include the following shared libraries:

- `FORTE_ROOT\install\lib\qqhd.`*xxx*
- `FORTE_ROOT\install\lib\qqsm.`*xxx*
- `FORTE_ROOT\install\lib\qqfo.`*xxx*
- `FORTE_ROOT\install\lib\qqdo.`*xxx*
- `FORTE_ROOT\install\lib\qqsh.`*xxx* (not on some UNIX platforms)
- `FORTE_ROOT\install\lib\qqcm.`*xxx*
- `FORTE_ROOT\install\lib\qqkn.`*xxx* (named qqknpthrd on some UNIX platforms)
- the directory containing the shared library file for the C++ API (.so or .a file, depending on the platform), which is installed in the `FORTE_ROOT/userapp` subdirectory containing the files for your client partition

xxx stands for different extensions, depending on the platform, as shown:

Platform	Shared Library Extension
Alpha OpenVMS	.exe
Alpha OSF/1	.so
AViion DG/UX	.so
HP 9000 HP/UX	.sl
RS/6000 AIX	.a
Sequent DYNIX/ptx	.so
VAX OpenVMS	.exe

Writing a C++ Client Application

This section describes how you can write a C++ client application using the generated C++ handle classes for the iPlanet UDS runtime system and your iPlanet UDS client partition.

In general, your C++ client application can perform functions independently, then start an iPlanet UDS client partition in the same process, interact with the iPlanet UDS application using generated handle classes and methods, then stop the iPlanet UDS client partition when the C++ client application has finished with it.

Because iPlanet UDS can take considerable resources to start up and shut down, we recommend that you start the iPlanet UDS client partition once and leave it running as long as your C++ client application needs it.

The C++ application can also use multiple threads and have these threads interact with iPlanet UDS tasks independently.

How to Use qqhTaskHandle

When you invoke a method on a handle class, you need to pass the reference to a running iPlanet UDS task as the first parameter of all methods and function calls except a few global functions. The reference to a running iPlanet UDS task is a qqhTaskHandle object, which is returned by the ForteStartup( ) global function when you start an iPlanet UDS task, as described in the next section. The following example shows a typical method call, where gTask1 is a qqhTaskhandle object:

```
currAccount = BankServerSO.GetAcctData(gTask, acctNumber);
```

How to Use iPlanet UDS Data Types

Certain methods generated as part of the C++ API specify some of their parameters or return values using iPlanet UDS data types.

The following table explains these data types:

Type in C++ API	TOOL type	C++ equivalent
qqos_bool	boolean	unsigned char
qqos_double	double	double
qqos_float	float	float
qqos_i1	char	char (1-byte integer)
qqos_i2	i2	short (2-byte integer)
qqos_i4	i4	long int (4-byte integer)
qqos_pointer	pointer	void *
qqos_ui1	char	unsigned char (1-byte integer)
qqos_ui2	ui2	unsigned short (2-byte integer)
qqos_ui4	ui4	unsigned long (4-byte integer)

The TOOL data types are described in the *TOOL Reference Guide*.

If you are writing a C++ application intended for a single platform, you can use the platform's equivalent data type in your C++ code. For example, use a long in place of qqos_i4.

However, if you intend to have your C++ application run on multiple platforms, we recommend that you use the data types defined by iPlanet UDS, which are portable across the platforms supported by iPlanet UDS.

NOTE In TOOL, if a parameter is defined as a TextData object, you can usually substitute a string value. This automatic conversion does not work for the C++ API. If a member function of the C++ API requires a qqhTextData object as a parameter, you must use a qqhTextData object.

If you receive a qqhTextData object, and wish to use the value of the object as a string, use the AsCharPtr member function of qqhTextData to convert the value to a string.

Start iPlanet UDS Interaction

The first action your C++ code must perform is to define an instance of the handle class for a task handle, qqhTaskHandle. The next step is to use the global function ForteStartup to start an iPlanet UDS client partition that runs in the same process as the C++ client application. ForteStartup starts an iPlanet UDS task associated with this C++ process and returns a handle to that task. This task handle represents the main task accessible in the iPlanet UDS client partition.

The following example shows how to define a task handle and start a task in an iPlanet UDS client partition:

```
extern qqhTaskHandle ForteStartup();
qqhTaskHandle gTask;
...
int main(int argc, char** argv)
{
  printf(
    "Starting the C++ client to the BankServer service
object!\n");
  gTask = ForteStartup();
  ...
}
```

File: cppbancl.cpp

For more information about the ForteStartup global function, see "ForteStartup Function" on page 298. For more information about task handles, see the iPlanet UDS online Help.

Passing Startup Parameters to iPlanet UDS

You can pass start-up parameters to iPlanet UDS by passing the parameters using the ForteStartup function.

The ForteStartup function has the following signatures:

qqEXPORTFUNCTION (qqhTaskhandle) ForteStartup();

qqEXPORTFUNCTION (qqhTaskHandle) ForteStartup(int *argc*, char* *argv*[]);

You can use the first signature of the ForteStartup function if you do not want to specify any iPlanet UDS startup parameters.

You can use the second signature of the ForteStartup function to specify iPlanet UDS start-up parameters.

The `argc` and `argv` parameters are similar to those for the C++ main(int argc, char *argv[]) function.

The `argc` parameter specifies the number of parameters in the array of strings passed by the argv parameter.

The `argv` parameter specifies an array of strings that each contain a word of the string of start-up parameters you want iPlanet UDS to use when your C++ application starts an iPlanet UDS client.

For example, suppose you want to specify start-up flags for the iPlanet UDS client partition. If you were specifying the ftexec command-line equivalent, you would write:

```
ftexec -fl %stdout(trc:user err:user) -fns mimi:5000
```

Similarly, you could specify these start-up parameters as the ForteStartup parameters, as shown:

```
qqhTaskHandle gTask;
int num_parms = 5;
char* parms [] = {"cppbancl", "-fl","%stdout(trc:user err:user)",
  "-fm", "(n:2000,x:5000");
gTask = ForteStartup(num_parms, parms);
```

NOTE The first element of the argv array (at argv[0]) should be the name of the executable for your C++ client application, which is cppbancl for the CppBanking example.

Logging Information for iPlanet UDS Client Partitions

The logging information for the iPlanet UDS client partition is written to a file in the `FORTE_ROOT/log` directory. The name of the file is:

`application_ID_process_ID.`**`log`**

An example log file name is `cppbanki_382.log`.

Interacting with Service Objects

The C++ API header file contains global functions that return handles to service objects. These global functions have the same names as the service objects.

Because service objects are usually accessed by several methods, you should define a C++ global variable to reference the service object. To define a global variable, define the variable for the handle to the service object outside any functions.

If you reference a service object in a partition that is not yet running, iPlanet UDS auto-starts the partition.

The following example shows how you can get a reference to a service object, then invoke methods on the service object:

```
extern qqhTaskHandle ForteStartup();
qqhBankService gBankServer;
qqhTaskHandle gTask;
qqhBankService BankServerSO;
...
```

```
int main(int argc, char** argv)
{
  gTask = ForteStartup();
// Use a global function defined in the .h file to get a reference
// to the service object.
  gBankServer = BankServer();
  ...
}
```

File: cppbancl.cpp

Using Handle Classes and Methods

When you write a C++ client application that uses the C++ APIs for the iPlanet UDS applications or runtime system, you can use the handle classes defined in the header files to create and destroy instances of iPlanet UDS objects.

For a detailed description of how iPlanet UDS classes, methods, attributes, and service objects are defined in the C++ API, see "Elements of the C++ API to a Client Application" on page 289.

Creating new iPlanet UDS objects: New( ) To create an instance of the handle class and its corresponding iPlanet UDS object, you need to use the New( ) member function, which is defined on the qqhObject handle class and inherited by all handle classes, as shown in the following example:

```
qqhBankAccount currAccount;
currAccount.New(gTask1);
```

Alternatively, you can assign the handle object to the reference for an existing iPlanet UDS object.

NOTE If you try to invoke a function call on the handle object before invoking the New member function on the handle object or assigning the handle object the reference to an existing iPlanet UDS object, you will get a NIL object runtime error.

For more information about the New( ) member function, see "New( ) Member Function" on page 301.

Calling methods using member functions After you have created or assigned an iPlanet UDS object to a new variable of a given handle classes, you can call member functions on this object, just as you would call the methods if you were using TOOL.

Setting attributes iPlanet UDS translates class attributes to a pair of get and set member functions, as shown in the following example. These member functions let you get and set the AcctBalance attribute on the BankAccount object:

```
double AcctBalance(const qqhTaskHandle& task);
void AcctBalance(const qqhTaskHandle& task, const double & value);
```

Deleting references to iPlanet UDS objects: Delete( ) The Delete( ) member function deletes the reference to the iPlanet UDS object held by the handle object. You usually do not need to delete this reference explicitly, because these references are automatically deallocated when they go out of scope in the C++ function.

To delete a reference to an iPlanet UDS object, use the Delete member function, which is defined on the qqhObject handle class and is inherited by all handle classes.

The Delete member function does not release the memory for the iPlanet UDS object itself or for the handle object. This member function is the equivalent of setting an iPlanet UDS object reference to NIL so that the iPlanet UDS memory reclamation function (garbage collection) releases the memory for the iPlanet UDS object. The handle class object itself is deallocated when the object goes out of scope. The qqhObject.Delete member function is also described in "Delete( ) Member Function" on page 300.

The following example shows how you can invoke a method on the BankServerSO service object to get a BankAccount object, then retrieve an attribute of that BankAccount object and set another attribute of that object.:

```
...
int main(int argc, char** argv)
{
  ...
  qqhBankAccount currAccount;
  // Get the account based on the account number.
  currAccount = BankServerSO.GetAcctData(gTask1, acctNumber);
  double currBalance;
  // Get the account balance.
  currBalance = currAccount.AcctBalance(gTask1);
  // Change the name of the owner on the account.
  qqhTextData acctOwner;
  qqhTextData acctOwner.new(gTask1);
  acctOwner.SetValue('Greta Garbo');
  currAccount.AcctName(gTask1, acctOwner);
...
}
```

File: cppbancl.cpp

Interacting with the iPlanet UDS Runtime System

iPlanet UDS provides handle classes for all supplier plans to the main project for the iPlanet UDS client application. Therefore, iPlanet UDS generates handle classes and member functions for all the classes in the Framework library, including classes that enable you to interact with the iPlanet UDS runtime system.

"Interacting with the iPlanet UDS Runtime System" on page 283 discusses these concepts in more detail.

Shutting Down the iPlanet UDS Client Partition

When your C++ client application has finished using iPlanet UDS, you can use the ForteShutdown global function to shut down the iPlanet UDS client partition.

When you shut down the iPlanet UDS client partition, all transient data used by the partition is deallocated.

iPlanet UDS server partitions that have been started by the C++ client application stay running after the ForteShutdown member function runs, just as they would after any other iPlanet UDS client partition shuts down. For more information about the ForteShutdown global function, see "ForteShutdown Function" on page 299.

The ForteShutdown( ) member function automatically dereferences the iPlanet UDS task, so that the memory for the client partition will be released.

The following example shows how you should shutdown the iPlanet UDS client partition after your C++ client application has finished using iPlanet UDS:

```
int StopForte()
{
// Perform clean up functions.
...
  ForteShutdown(gTask1);
}
```

Handling iPlanet UDS Exceptions

iPlanet UDS provides the following macros, which you can use in your C++ client application to catch iPlanet UDS exceptions:

qqhTRY(*task*) starts a try block

qqhCATCH(*task*, *class*, *var*) starts a catch block, which catches iPlanet UDS exceptions of the specified handle class

qqhELSE_CATCH(*task*, *class*, *var*) statement in a catch block that catches iPlanet UDS exceptions of the specified handle class

qqhELSE(*task*) statement in a catch block that catches unexpected iPlanet UDS exceptions of the specified handle class

qqhELSE_ONLY(*task*) catches all iPlanet UDS exceptions, without being part of a catch block

qqhEND_TRY(*task*) ends a try block

The following example shows the general structure for the try and catch statements:

```
qqhTRY(task)
... code interacting with the iPlanet UDS application or runtime system
qqhCATCH(task, class, var)
... code handling the exception of the specified class
qqhELSE_CATCH(task, class, var)
... code handling the exception of the specified class
qqhELSE_CATCH(task, class, var)
... code handling the exception of the specified class
... any other qqhELSE_CATCH statements
qqhELSE(task)
... code handling any other iPlanet UDS exceptions
qqhEND_TRY(task)
```

You can use the qqhELSE_ONLY macro to catch any iPlanet UDS exceptions, without first defining a catch block, as shown:

```
qqhTRY(task)
... code interacting with the iPlanet UDS application or runtime system
qqhELSE_ONLY(task)
... code handling any raised iPlanet UDS exceptions
qqhEND_TRY(task)
```

These macros are discussed in more detail in "Exceptions" on page 294.

The following example, shows how you could use the qqhTRY, qqhCATCH, and qqhELSE_CATCH macros provided by iPlanet UDS to catch iPlanet UDS exceptions:

```
void AccountLoop()
  {
    printf("Enter an account number (or '0' to quit): ");
    qqos_i4 selAcct;
    int result = scanf("%ld", &selAcct);
    if(selAcct == 0)
      return;
    qqhTRY(gTask)
    {
```

File: cppbancl.cpp

```
      gCurrentAccount = gBankServer.GetAcctData(gTask, selAcct);
    }
    qqhCATCH(qqhAccountNotFound, unknownExcept)
    {
      printf("An qqhAccountNotFound occurred in
AccountLoop().\n");
    }
    qqhELSE_CATCH(qqhGenericException, unknownExcept)
    }
    qqhEND_TRY;
    // Go to transaction menu for the account.
    ...
    }
  }
}
```

NOTE You can nest iPlanet UDS qqhTRY blocks within C++ TRY blocks; however, you cannot overlap the qqhTRY blocks and TRY blocks, and you cannot nest C++ TRY blocks within iPlanet UDS qqhTRY blocks.

Compiling the C++ Client Application

For information about specific compiling and linking options for each platform, see iPlanet UDS Technote 10947.

Deploying the C++ Client Application

To deploy the C++ client application on client machines, you need to install the following:

- iPlanet UDS runtime system
- shared library file for the C++ API (.dll, .so, or .a file, depending on the platform)
- .exe file for the C++ client application

Interacting with the iPlanet UDS Runtime System

This section explains the steps for writing a C++ client application that interacts with the iPlanet UDS runtime system. These steps will be slightly different for each platform.

This section also outlines the capabilities and restrictions that you need to be aware of to write a C++ client application that interacts with the iPlanet UDS runtime system.

iPlanet UDS automatically provides C++ APIs—header files and shared libraries—for all the classes and runtime objects defined as iPlanet UDS libraries. You can locate the definitions for this C++ API by looking at the .txt files in the `FORTE_ROOT/install/inc/handles` directory. These files describe what .hdg files in that directory contain the class definitions for the handle classes that represent iPlanet UDS classes. For more information about these C++ APIs, see "The C++ API to the iPlanet UDS Runtime System" on page 302.

Working with iPlanet UDS Classes

iPlanet UDS automatically provides a C++ API—header files and shared libraries—for all the classes and runtime objects defined as iPlanet UDS libraries, except the Display library.

You can locate the definitions for this C++ API by looking at the .txt files in the `FORTE_ROOT/install/inc/handles` directory. This file describes what .hdg files in that directory contain the class definitions for the handle classes that represent iPlanet UDS classes and maps the iPlanet UDS classes to the files that contain their C++ handle class definitions.

NOTE The C++ class definitions for the handle classes can include classes, methods, or attributes that are not part of the documented and supported iPlanet UDS libraries. You should not use undocumented classes, methods, or attributes in the handle classes.

Working with iPlanet UDS Runtime Objects

Using the handle classes provided for the iPlanet UDS classes, you can access iPlanet UDS runtime objects, such as the object location manager, the distributed object manager, and the log manager.

To access these runtime objects, start with a task handle. For example, to access the log manager for a partition, you first need to get a handle for the partition, then a handle to the log manager, as shown in the following example:

```
qqhTaskHandle task;
task = ForteStartup();
qqhPartition part;
part = task.Part();
qqhLogMgr logmgr = part.LogMgr();
logmgr.PutLine(source = 'Found the log manager.');
```

Chapter 15

C++ API Reference Information

This chapter describes the handle classes that are generated by iPlanet UDS when you have iPlanet UDS generate a C++ API for a client partition.

This chapter contains information about the following topics:

- the files that make up the C++ API
- the standard format of the generated handle classes, handle methods, and the global functions that access service objects
- additional methods that are added to handle classes that represent iPlanet UDS classes
- guidelines for how to use the handle classes, handle methods, and global functions that represent service objects in TOOL

Files Generated as Part of a C++ API

iPlanet UDS generates the following files as part of the C++ API. These files are all critical for using the C++ API:

- *client_component_id*.txt
- *client_component_id*.h
- *client_component_id.xxx* (shared libraries)
- *client_component_id*.lib (Windows NT and Windows 95 only)
- c1.cdf, c2.cdf, and so forth
- p1.h, p2.h, and so forth

The steps for generating these files for the C++ API are described in "Generating a C++ API for an iPlanet UDS Application" on page 259.

client_component_id.txt

The *client_component_id*.txt file generated for the C++ API contains information about:

- TOOL classes that are accessible using the API
- handle classes that you use to interact with the TOOL classes
- files that contain the class definitions for the handle classes (c#.cdf files)
- names for the C++ classes that directly represent the TOOL classes
- service objects accessible using the C++ API

You can generate a C++ API for a partition that uses classes and subclasses from the Display library; however, you can only access classes that are not classes and subclasses of the Display library. Therefore, these classes are not included in the *client_component_id*.txt file.

The following example shows a generated *client_component_id*.txt file:

```
Readme for C++ API for partition: cppban0

This file describes the files and classes that
have been generated for using this C++ API.
Project: BankServices

TOOL Class        File Name  Handle Class        C++ Class Name
-----------------------------------------------------------------
AccountNotFound   c3.cdf     qqhAccountNotFound  AccountNotFound_c3
BankAccount       c4.cdf     qqhBankAccount      BankAccount_c4
BankService       c5.cdf     qqhBankService      BankService_c5

Project: CppBanking

TOOL Class        File Name  Handle Class        C++ Class Name
-----------------------------------------------------------------
CppAPI            c6.cdf     qqhCppAPI           CppAPI_c6

Service Objects    Class            Function to retrieve service object
-----------------------------------------------------------------
BankServer         qqhBankService   BankServer();
```

You can use the *client_component_id*.txt file to understand what handle classes (and corresponding TOOL classes) can be accessed by C++ clients.

Some of the c#.cdf files are used only by iPlanet UDS, and are therefore not listed in the *client_component_id*.txt file for this C++ API. Do not use classes defined in files not listed in the *client_component_id*.txt file.

client_component_id.h

The *client_component_id*.h file is the main header file for the C++ API. This file includes all the header files that this API requires, except for the c#.cdf files that are described in the *client_component_id*.txt file.

You need to include this file (#include) in your C++ client application, as shown in the following example:

```
#include Bankin0.h
```

This file also contains global functions that return handles to iPlanet UDS service objects that the C++ API can access.

For example, the following example shows the global function defined for the BankServices.BankServer service object:

```
qqEXPORTFUNCTION(qqhBankService) BankServer();
```

client_component_id.xxx (shared library)

This file is the compiled shared library file for the C++ API, and the extension, indicated above as xxx, depends on the platform, as shown:

Platform	Shared Library Extension
Alpha OpenVMS	.exe
Alpha OSF/1	.so
AViion DG/UX	.so
HP 9000 HP/UX	.sl
RS/6000 AIX	.a
Sequent DYNIX/ptx	.so
VAX OpenVMS	.exe
Windows 95	.dll and .lib
Windows NT	.dll and .lib

The handle classes and member functions provided by this shared library are described in the *client_component_id*.txt file.

This file needs to be in a directory specified by the library search path on your system. On UNIX and the Macintosh platforms, you need this shared library file when you build your C++ client application. On all platforms, this .dll file is part of the deployment for your C++ client application.

client_component_id.lib

This file is generated on Windows NT and Windows 95 only. This file describes the contents of the corresponding .dll file.

On Windows NT and Windows 95, you need to specify this file when you link your application. You do not need this file at runtime.

c#.cdf

The c#.cdf files contain the class definitions for the handle classes and C++ classes that represent the TOOL classes. One .cdf file is generated for each TOOL class. (.cdf stands for class definition file.)

The class definition for the handle class is usually in the second half of the file. The class statement for the handle class contains "`class qqEXPORTCLASS qqh`*`classname`*".

The following example shows the class definition for a handle class in a c#.cdf file:

```
class qqEXPORTCLASS qqhBankService : public qqhObject
{
public:
  qqhBankService();
  qqhBankService(const qqhBankService& other);
  qqhBankService(BankService_c5*);
  ~qqhBankService();
  qqhBankService& operator=(const qqhBankService& other);
  operator BankService_c5*() const;
  void New(const qqhTaskHandle& task);

  // Attribute Get/Set pairs
  qqhArray AcctList(const qqhTaskHandle& task);
  void AcctList(const qqhTaskHandle& task, const qqhArray& value);
```

```
  // Methods

  void Init(const qqhTaskHandle& task);
  double UpdateAcct(
    const qqhTaskHandle& task, qqos_i4 p1, double p2);
  qqhBankAccount GetAcctData(const qqhTaskHandle& task, qqos_i4 p1);
  qqhArray GetAcctNumList(const qqhTaskHandle& task);
};
```

You need to include each c#.cdf file that contains the class definition for a handle class you are using in your C++ client application.

Some of the c#.cdf files are used only by iPlanet UDS, and are therefore not listed in the *client_component_id*.txt file for this C++ API. Do not use classes defined in files not listed in the *client_component_id*.txt file; the behavior of these classes is undefined.

p#.h

The p#.h files are used only by iPlanet UDS, and are therefore not listed in the *client_component_id*.txt file for this C++ API. These files are included in the *application_name*.h file and need to be in the include directory when you build your C++ client application.

Elements of the C++ API to a Client Application

This section describes the structure and conventions that iPlanet UDS uses to generate a C++ API interface from the client partition of an iPlanet UDS application. This section describes how iPlanet UDS translates the following elements defined in TOOL when it generates the interface header file for C++:

- classes
- methods
- attributes
- service objects
- exceptions
- events

Handle Classes

For each class defined in the main project of a client application and in its supplier plans, iPlanet UDS produces a handle class. These handle classes represent the classes defined in the project that was the main project of the application, and all classes from the supplier plans that are used by the application.

The handle classes generated as part of the C++ API use the following naming convention:

qqh[*plan*_**]***class*

plan is the name of the iPlanet UDS plan (project, library, or model) that defines the class corresponding to the handle class. This part is only used if the class name is not unique within the iPlanet UDS client application.

class is the name of the iPlanet UDS class corresponding to the handle class.

For example, a handle class corresponding to the Branches class in the BankServices project would be named qqhBranches. However, if two different supplier projects, Accounting and Personnel, define classes named Record, then the handle classes for these classes would be qqhAccounting_Record and qqhPersonnel_Record.

These handle classes and their methods allow you to write C++ applications that interact with iPlanet UDS objects and runtime processes without having to worry about the internals of how iPlanet UDS manages objects. For example, the handle classes ensure that you can write C++ code that interacts with iPlanet UDS objects without needing to know:

- where a distributed object is located
- how and when memory is allocated and deallocated when you create and destroy iPlanet UDS objects
- where objects are stored when iPlanet UDS reclaims memory using its garbage collection facilities

These handle classes and their methods manage these kinds of issues as part of interacting with the TOOL methods that they represent.

C++ Classes—for Type Conversion

iPlanet UDS generates a C++ class that represents the same class as the handle class. This class, however, is only to be used to cast an object to a subclass. This class is shown in the C++ Class Name column of the .txt file for the C++ API. The name of this class has the format:

*TOOL_class_name*_**c#**

For example, the C++ class name for the BankAccount class is BankAccount_c4. If you needed to cast a qqhObject object to an BankAccount object, you would need to use the following syntax:

```
// qqhArray.FindObjectForRow returns qqhObjects, which we need to
// convert to qqhBankAccount objects.
  qqhBankAccount* currAcct = new
    qqhBankAccount((BankAccount_c4*)((qqlo_Object*)curObj));
// Print out the account information.
PrintAccountInfo(*currAcct);
```

File: cppbancl.cpp

Methods

Within each handle class, iPlanet UDS generates a member function corresponding to each method in the TOOL class represented by the handle class.

These member functions have the same name as the original method. For example, the Framework library class DateTimeData has a method named SetCurrent. The corresponding member function in the qqhDateTimeData class is also named SetCurrent.

The member function has the same name as its iPlanet UDS counterpart and its signature is the C++ equivalent of the signature for the TOOL method. All member function signatures define the first parameter as the handle for the iPlanet UDS task under which this method is invoked. The following examples compare a TOOL method signature with the signature for its counterpart C++ member function:

Code Example 15-1 TOOL method

```
BankService.GetAcctData(input AcctNumber: Framework.integer):
   copy BankServices.BankAccount
```

Code Example 15-2 C++ member function

```
qqhBankAccount GetAcctData(const qqhTaskHandle& task,
  qqos_i4 AcctNumber);
```

The parameters of the TOOL methods are translated to their C++ equivalents as shown in the following table:

TOOL	C++	Description
input	**const** *type***&**	No change to the value of the reference. The contents of the referenced object might change.
output	*type***&**	Expect the reference to a new object to be returned.
input output	*type***&**	Expect the referenced object to change.
copy input	**const** *type***&**	No change to the value of the reference and no change to the referenced object.
copy output	*type***&**	Expect the reference to a copy of a new object to be returned.
copy input output	*type***&**	Expect the input object to be copied and changed, and a reference to a copy of the changed object to be returned.

Attributes

Within each handle class, iPlanet UDS generates two member functions that correspond to each attribute, one that returns the value of the attribute, and another that sets the value of the attribute.

iPlanet UDS does not include virtual attributes or their methods in the generated C++ handle classes.

For example, the Framework library class DateTimeData has an attribute called HoursFromGMT. iPlanet UDS generates two member functions for the qqhDateTimeData class that have the following signatures:

```
//Attribute Get/Set pairs
int HoursFromGMT(const qqhTaskHandle& task);
void HoursFromGMT(const qqhTaskHandle& task, const int& value);
```

Service Objects

Any service objects that are defined in the main project of the iPlanet UDS client application or that are accessed by the iPlanet UDS client application in its supplier plans can be initialized by using a generated global function.

In the main C++ API header file, iPlanet UDS generates a function like the following for each defined service object:

```
qqEXPORTFUNCTION(qqhClock) ClockService();
```

The name of the function corresponds to the name of the service object. The function returns a service object.

qqEXPORTFUNCTION indicates that this function is available to other applications, such as a C++ client application. Before you use this function in your C++ code, you need to declare the function using the extern keyword, as shown in the following example:

```
extern qqhBankService BankServer();
```

To initialize this service object, you need to write C++ code like the following:

```
qqhClock ClockService = ClockService();
```

Exceptions

You can handle exceptions that have been raised by the iPlanet UDS methods that are called by your C++ application by using the following macros that are provided by iPlanet UDS:

Macro	Description
qqhTRY(*task*)	Opens an exception handler (try) block.
qqhCATCH(*class*, *var*)	Defines a block for handling all exceptions of a specific type.
qqhELSE_CATCH(*class*, *var*)	Adds a handler for an additional exception type.
qqhELSE()	Defines a block for handling an exception of an unexpected type.
qqhELSE_ONLY*()*	Define a block that catches all exceptions. Does not need to follow a qqhCATCH statement.
qqhEND_TRY()	Closes an exception handler block.

These macros specify the following parameters:

Parameter	Description
task	Reference to task handle for the iPlanet UDS task that might raise the specified exception.
class	Name of handle class corresponding to the iPlanet UDS exception class that this statement can catch.
var	Variable that will be the name of the handle object that references the iPlanet UDS exception, when the exception is raised and caught.

When a qqhCATCH or qqhELSE_CATCH statement catches an iPlanet UDS exception, the macro creates an instance of the handle class specified in the `class` parameter with the name specified in the var parameter. You can then use this reference to get information from the attributes of the raised iPlanet UDS exception.

We recommend that you include a try block around all calls to iPlanet UDS methods, as shown in the following example, which shows how you can catch exceptions much the way you would in TOOL. This example also demonstrates how you can re-throw the iPlanet UDS exception as a C++ exception:

```
qqhTRY(gTask)
{
  // Update the account balance.
  acctBalance =
  gBankServer.UpdateAcct(gTask, acctNum, transAmt);
}
qqhCATCH(qqhAccountNotFound, noAccountFound)
{
  printf("No account with number %ld was found.\n\n", acctNum);
}
qqhELSE_CATCH(qqhGenericException, unknownExcept)
{
  printf("A GenericException occurred in AccountLoop().\n");
  qqhTextData msgTextData = unknownExcept.GetMessage(gTask);
    printf("message is %s\n", msgTextData.AsCharPtr(gTask));
  // Rethrow the exception as a C++ exception.
  throw(Forte_UnexpectedException(
    "A GenericException occurred in UpdateAccount()."));
  }
qqhEND_TRY;
```

File: cppbancl.cpp

Alternatively, you can use the qqhELSE_ONLY macro to catch any raised exception, as shown in the following example:

```
// The task is the current qqhTaskHandle.
qqhTRY(task)
{
  int result = gCustomerMgr.RemoveDuplicates(task);
  printf("%d Duplicate customer records removed.\n", result);
}
qqhELSE_ONLY(task)
{
  printf(
     "iPlanet UDS exception thrown by gCustomerMgr.RemoveDuplicates.\n");
}
```

Events

iPlanet UDS does not provide any way for C++ code to either explicitly post events or to register for events when they are posted by the iPlanet UDS application.

If a method on a handle class posts an event, you can call that method within your C++ application and thereby post an event.

If you want your C++ application to respond to iPlanet UDS events that are posted by the application or the iPlanet UDS runtime system, you should define an event loop in a TOOL method that waits for that event. You can then have a thread of your C++ application start a task that calls that method and waits in that event loop until the event is posted. The method can then return a value or an object to the C++ application to supply information about the posted event.

If you like, you can define an event loop in a method that runs in the C++ client partition for which the C++ API is generated. The C++ client application can then invoke the method on the client partition to indirectly register and wait for an event.

You can also define a class with an attribute that indicates that an event has occurred. You can then start an event loop to listen for a specific event. When the loop catches the event, the TOOL code then changes the value of the attribute on the object. In the meantime, you can define a C++ listener task, which checks the attribute of the service object at intervals to see whether the attribute value has changed.

Special Handling for Array and Pointer to Char Parameters

Certain iPlanet UDS data types produce the same C++ declarations because iPlanet UDS is a more strongly typed language than C++. For example, in iPlanet UDS, an Array of TextData, an Array of Object, and an Array of DateTimeData are considered different data types. However, these types all map to the qqhArray handle class. Similarly, a pointer to char data type and the iPlanet UDS String data type are both mapped to char*. These mappings would cause problems in the case of overloaded iPlanet UDS methods, because the methods would have identical parameter lists when translated into C++ methods.

To ensure that method signatures are unique in C++, iPlanet UDS generates an additional dummy parameter at the end of the parameter list for Array data types and pointer to char data types.

The dummy parameter data type is defined as "___M_ccc_mmm *", where:

Part	Description
___M	Three underscores and a capital M
ccc and mmm	Two numbers uniquely identifying the method

See the .cdf file for the handle class for the exact name of the parameter type.

When you invoke a method that has a dummy parameter, you must assign a NULL value cast to the data type for the dummy parameter, as shown in the following example, where ___M_6_3 * is the data type of the dummy parameter:

```
myObject.getData(gTask, currArray, (___M_6_3 *) NULL);
```

Arrays For example, suppose your iPlanet UDS application defines the following method signatures to overload the getData method with different types of Arrays as parameters:

```
getData(input output data : Array of TextData)
getData(input output data : Array of Object)
```

iPlanet UDS generates the following C++ signatures for these methods:

```
void getData (const qqhTaskHandle& task, qqhArray& data, ___M_6_3 *p2)
void getData (const qqhTaskHandle& task, qqhArray& data, ___M_6_4 *p2)
```

pointer to char Similarly, suppose your iPlanet UDS application defines the following method signatures to overload the setData method with a string and a pointer to char as the different parameters:

```
myClass.setData(data : string)
myClass.setData(data : pointer to char)
```

iPlanet UDS generates the following C++ signatures for these methods:

```
void setData (const qqhTaskHandle& task, char *data)

void setData (const qqhTaskHandle& task, char *data, ___M_7_9 *p2)
```

Utility Global Functions and Member Functions

This section describes the global functions and member functions that enable you to write C++ applications that interact with iPlanet UDS objects and the runtime system.

iPlanet UDS generates functions for starting and shutting down the iPlanet UDS runtime system. iPlanet UDS also generates additional member functions for the handle classes that correspond to the Object and TaskHandle classes belonging to the Framework library.

Functions that Start and Stop the iPlanet UDS Runtime System

iPlanet UDS automatically generates the following functions as part of the C++ API, which let you start and shut down the iPlanet UDS runtime system:

ForteStartup Starts up the iPlanet UDS runtime system for the C++ API.

ForteShutdown Shuts down the iPlanet UDS runtime system for the C++ API.

ForteStartup Function

This function starts up the iPlanet UDS runtime system for the C++ API. When this function completes, it sets its qqhTaskHandle object to reference the task handle for the iPlanet UDS runtime system.

This function has the following signature:

qqEXPORTFUNCTION(qqhTaskHandle) ForteStartup( );

You must explicitly declare that the prototype for ForteStartup( ) is defined elsewhere, by specifying the function prototype with the extern keyword as a global name in your C++ application.

The following example shows how you can start an iPlanet UDS client partition within your C++ code:

```
extern qqhTaskHandle ForteStartup();
qqhTaskHandle gTask;
...
int main(int argc, char** argv)
{
  gTask = ForteStartup();
...
}
```

To shut down the runtime system for the C++ API, use the ForteShutdown function, described in the following section.

ForteShutdown Function

This function shuts down the iPlanet UDS runtime system for the C++ API. You typically use this function in the cleanup code when shutting down an application.

void ForteShutdown(qqhTaskHandle&);

The ForteShutdown( ) member function automatically dereferences the iPlanet UDS task, so that the memory for the client partition will be released.

The following example shows how you can shut down the iPlanet UDS runtime system for the C++ API within your C++ code:

```
int ShutDown()
{
  ForteShutdown(gTask);
...
}
```

To start the runtime system for the C++ API, use the ForteStartup function, described in "ForteStartup Function" on page 298.

qqhObject Handle Class

The qqhObject class is the handle class for the Object class in the Framework library, and is the root class of the handle class hierarchy. iPlanet UDS generates member functions for qqhObject that represent all the methods documented for the TOOL Object class in the iPlanet UDS online Help.

iPlanet UDS generates four extra public member functions for the qqhObject handle class:

Delete( ) Deletes the reference to the iPlanet UDS object but does not delete either the handle object or the iPlanet UDS object itself.

IsNil( ) Checks whether the reference to the iPlanet UDS object is NIL.

New( ) Instantiates the handle object and the iPlanet UDS object, then assigns the reference to the iPlanet UDS object to the handle object.

SetObject(0) Sets the reference to the iPlanet UDS object to NIL.

Because all handle classes are derived from the qqhObject class, these public member functions are inherited by all handle classes.

Delete( ) Member Function

This member function deletes the reference to the iPlanet UDS object held by the handle object. This member function has the following signature:

void Delete(qqhTaskHandle& task);

The Delete( ) member function deletes the reference to the iPlanet UDS object held by the handle object. You usually do not need to delete this reference explicitly, because these references are automatically deallocated when they go out of scope in the C++ function.

The Delete member function does not release the memory for the iPlanet UDS object itself or for the handle object. This member function is the equivalent of setting an iPlanet UDS object reference to NIL so that the iPlanet UDS memory reclamation function (garbage collection) releases the memory for the iPlanet UDS object. The handle class object itself is deallocated when the object goes out of scope.

The following example shows how you can delete an iPlanet UDS object within your C++ code:

```
// Delete a BankAccount object.
currAccount.Delete(gtask);
```

IsNil( ) Member Function

This member function checks whether the reference to the iPlanet UDS object is NIL. This member function has the following signature:

int IsNil( );

This member function returns 1 if the reference to the iPlanet UDS object is NIL, and 0 if it is not NIL.

New( ) Member Function

This member function creates a new iPlanet UDS object and makes it available to a C++ application. Alternatively, you can assign the handle object to the reference for an existing iPlanet UDS object.

This member function has the following signature:

void New(qqhTaskHandle& task);

The following example shows how you can create and access an iPlanet UDS object within your C++ code:

```
// Create a new BankAccount object and change values on it.
qqhBankAccount currAccount;
// Create the Forte object.
currAccount.New(gtask);
currAccount.Name(gtask, 'Greta Garbo');
```

NOTE If you try to invoke a function call on the handle object before invoking the New member function on the handle object or assigning the handle object the reference to an existing iPlanet UDS object, you will get a NIL object runtime error.

SetObject( ) Member Function

This member function sets the reference to the iPlanet UDS object to NIL. This member function has the following signature:

void SetObject(0);

You can only use this member function with a parameter of 0.

The C++ API to the iPlanet UDS Runtime System

iPlanet UDS provides access to the iPlanet UDS runtime system by providing C++ header files and shared library files that can be used when you write C++ client applications that interact with iPlanet UDS applications. You can interact with the iPlanet UDS runtime system by using handle classes defined and provided by iPlanet UDS and by using the classes and functions discussed in "Utility Global Functions and Member Functions" on page 298.

NOTE The C++ API is provided for all iPlanet UDS class libraries except the Display library.

Aside from the restrictions described in Chapter 14, "Accessing iPlanet UDS Using C++," you can use the handle classes to the TOOL classes to use the functions provided by these classes in TOOL. For information about how to use these classes, see the appropriate iPlanet UDS documentation:

The following table lists the iPlanet UDS libraries for which iPlanet UDS provides C++ APIs, along with the files that map the C++ handle classes to TOOL classes. For more information, see the iPlanet UDS online Help.

iPlanet UDS Library	File in FORTE_ROOT/install/inc/handlesdirectory
DDEProject	ddeproje.txt
Framework	framewor.txt
GenericDBMS	genericd.txt
OLE	ole2.txt
SystemMonitor	systemmo.txt

.hdg files Each .hdg file contains the class definitions for a C++ handle class that represents a TOOL library class. These files are installed in the `FORTE_ROOT/install/inc/handles` directory.

shared library files These files are used when linking the C++ client application. For a detailed list of these files, see "Setting up Your System and Compiler to Use the C++ API" on page 270. These files are installed in `FORTE_ROOT/install/lib`.

Part 5

Using Network and Operating System Features

Part 5 of *Integrating with External Systems* describes how you can use system activities and network sockets to enable your application to communicate with an iPlanet UDS applications.

Chapter 16, "Using System Activities and Network Connections"

Explains how to use the ActivityManager, SystemActivity, and Rendezvous classes to use system activities to communicate between external applications and iPlanet UDS applications.

It also explains how to use the ExternalConnections class to use network sockets to communicate between external applications and iPlanet UDS applications.

Chapter 16

Using System Activities and Network Connections

This chapter discusses how to interact with other applications using system activities and network connections.

For reference information about the TOOL classes that implement these features, see the iPlanet UDS online Help.

About Using System Activities and Network Connections

iPlanet UDS provides several classes that let you interact with external applications by coordinating system activities or by communicating using network features.

About System Activities

The Rendezvous, SystemActivity, and ActivityManager classes allow an iPlanet UDS application, C++ client applications, or wrapped C code to register for certain low-level operating system events, referred to as *system activities*. Your application can then wait for notification that an activity has completed, or occurred. The SystemActivity and ActivityManager classes do not support activities on Microsoft operating systems.

Detailed information about integrating using system activities is in "Using System Activities" on page 308.

About the ExternalConnection Class

The purpose of the ExternalConnection class is to facilitate network communications between two points (peer to peer or client to server) when one of the points is an iPlanet UDS application. Using the ExternalConnection class, an iPlanet UDS application can communicate with an external process or program that is running locally or on another host. For example, an iPlanet UDS application can exchange data with a Web browser, Java, C, or BASIC program, telnet, HTTP, and so on.

Detailed information about integrating using the ExternalConnection class and network sockets is in "Using the ExternalConnection Class" on page 316.

Using System Activities

iPlanet UDS includes classes that increase ways for iPlanet UDS applications to integrate with external systems. The `Rendezvous`, `SystemActivity`, and `ActivityManager` classes allow an iPlanet UDS application, C++ client applications, or wrapped C code to register for certain low-level operating system events, referred to as *system activities*. Your application can then wait for notification that an activity has completed, or occurred.

Supported System Activities

A program can register for the following system activities:

- user-defined activities
- interval timers (accessible from C and C++)
- UNIX signals
- UNIX file description events posted by the network or file system (data available, input channel available, exceptional data available)
- VMS asynchronous trap (AST) notifications for I/O or user data notifications
- VMS event flags

NOTE The `SystemActivity` and `ActivityManager` classes do not support activities on Microsoft operating systems. On Microsoft Windows 95 and Windows NT, you can write routines similar to those shown in this section by starting a thread and invoking Win32 API calls.

User-defined activities You can define TOOL methods, or C or C++ functions that define and post activities. For example, you could use these user-defined activities to notify iPlanet UDS when a series of activities outside the iPlanet UDS runtime system have occurred. You could also write a C function that posts a user-defined activity to return data to your iPlanet UDS application after it receives the data from another application. For more information about posting user-defined activities, see "Setting Up User-Defined Activities" on page 315.

Working with System Activities

This section discusses how to use the `ActivityManager`, `Rendezvous`, and `SystemActivity` classes to work with system activities.

Registering for Notification about System Activities

In TOOL code, you have a task register for iPlanet UDS events by writing an event loop, and including the events in a list of **when** clauses in the event loop. If your application does not register for an event, it does not receive the event, even when the event is posted.

Similarly, your application needs to register to receive notification about a system activity. You can use the `RegisterSystemActivity` methods on the `ActivityManager` class to register for particular system activities. The following example how you can register for a system activity:

```
sysActivity : SystemActivity =
    task.Part.OperatingSystem.ActivityManager.RegisterSystemActivity(
      systemId = fileId, activityType = SH_SA_FD_WRITE,
      userContext = pContext, rendezvous = myRendezvous);
```

The `RegisterSystemActivity` method returns a SystemActivity object, which represents a particular system activity for which a task has registered. The iPlanet UDS then notifies the Rendezvous object specified in the `RegisterSystemActivity` method when the system activity occurs.

NOTE If a task registers for the same system activity multiple times, the Rendezvous object will receive multiple notifications when the activity occurs.

In C++, you can register using the `RegisterSystemActivity` methods on the qqhActivityManager handle class.

In C, you can register for these system activities using the functions described in the iPlanet UDS online Help.

If you want your C or C++ code to use an interval timer, you can register to be notified about interval timer ticks in your C or C++ code. This interval timer is provided by iPlanet UDS to C and C++ clients, without requiring any iPlanet UDS Timer objects to be instantiated. In TOOL, you can instantiate a Timer object and register for the Timer.Tick event.

To register for an interval timer tick, you need to specify the tick interval as the systemID parameter value and use the `SH_SA_TIMER` constant (qqSH_SA_TIMER in C) for the `activityType` parameter. The following example shows how you can register for an interval timer tick activity in C:

```
me->SysAct = qqsh_RegisterSystemActivity((void *) interval,
(long) qqSH_SA_TIMER, (void *) me, rend);
```

In this example, me is a pointer to a struct, and rend is a pointer to a Rendezvous object.

Each time you register for an interval timer tick, iPlanet UDS starts a new interval timer, so you should deregister for these events as soon as you no longer need them.

Waiting for Activity Completion

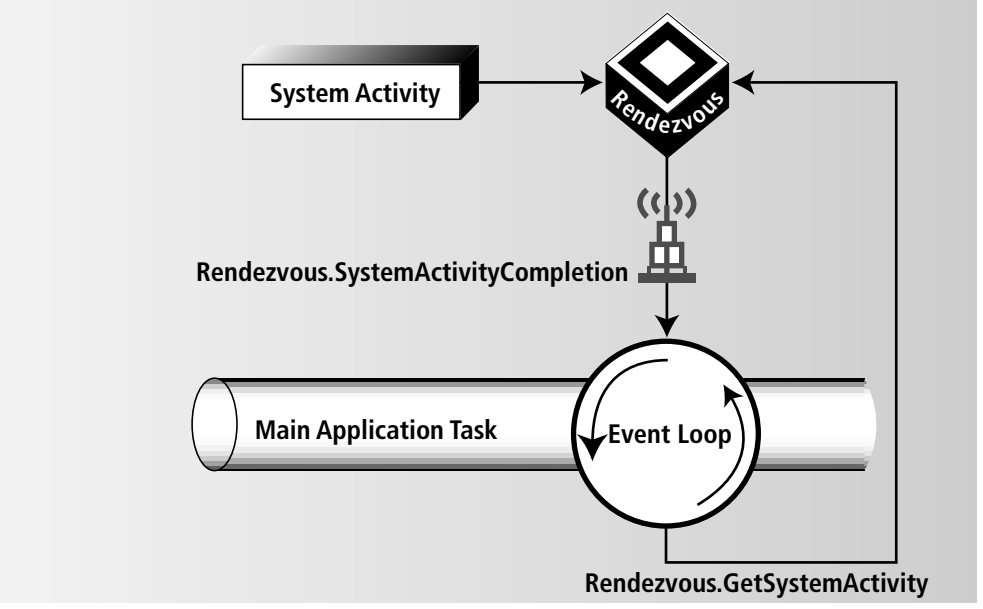
When a registered system activity occurs, the Rendezvous object is notified, and the system activity is added to a queue of system activities that can be accessed using the `Rendezvous.GetSystemActivity` method. Each invocation of the `GetSystemActivity` method returns a system activity from the Rendezvous object's queue, and removes that system activity from the queue.

Tasks can find out about completed system activities in the following ways:

Waiting for an event (TOOL only) The task registers for the Rendezvous.SystemActivityCompletion event in an event loop. This task can then wait for the SystemActivityCompletion event and any other events for which it has registered.

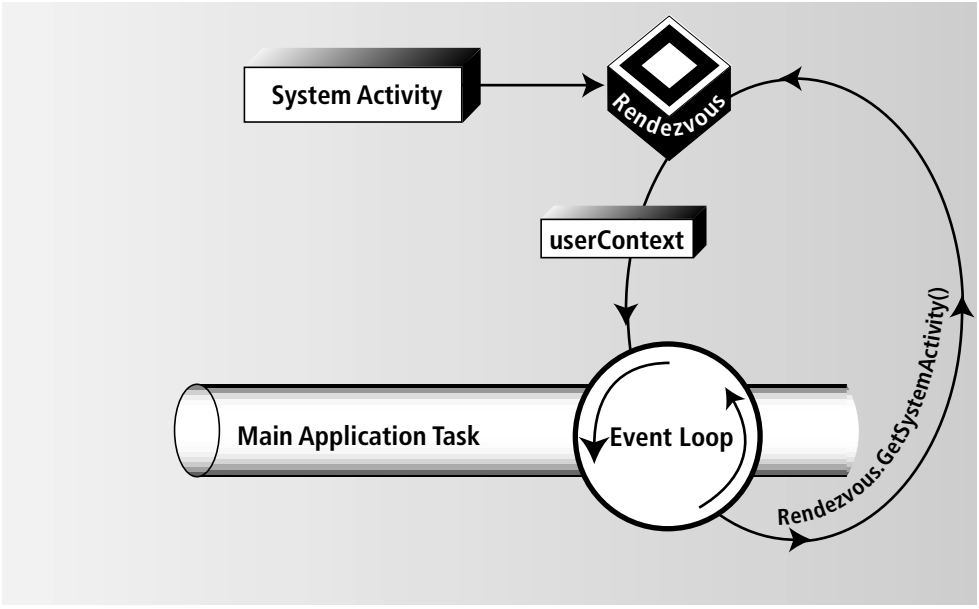
After the task catches the SystemActivityCompletion event, the task should invoke the `Rendezvous.GetSystemActivity` method to retrieve the system activity from the queue, as described in the next section, as shown in Figure 16-1:

Figure 16-1 Notification of a System Activity in TOOL with the SystemActivityCompletion event



Checking for system activities (TOOL, C++, and C) The task invokes the `GetSystemActivity` method on the Rendezvous object associated with a particular system activity, as shown in Figure 16-2:

Figure 16-2 Checking for System Activities Using the GetSystemActivity Method



If there are system activities in the Rendezvous object's queue, the method returns information about the oldest system activity in the queue, which is then removed from the queue.

If there are no system activities in the queue, the `GetSystemActivity` does one of the following, depending on the mode setting of the Rendezvous object:

- Poll

  If there are no system activities in the queue of the Rendezvous object, the `GetSystemActivity` method returns the value `-1`. After a user-defined interval, the task can invoke `GetSystemActivity` method again to check for system activities.

  Polling is the default mode, and is set for each Rendezvous object using the `Rendezvous.SetMode` method, which is described in the iPlanet UDS online Help.

- Block

  If there are no system activities in the queue of the Rendezvous object, the task waits in the method for a system activity to be added to the queue in the Rendezvous object. The method does not return until it has found a system activity.

To set the Rendezvous object's mode to block, use the `Rendezvous.SetMode` method, described in the iPlanet UDS online Help.

NOTE If a task will block while waiting for the SystemActivityCompletion event, the task should register for the SystemActivityCompletion event before invoking the `RegisterSystemActivity` method. Because iPlanet UDS events in an event loop are registered before any TOOL code within the event loop is executed, you can simply invoke the `RegisterSystemActivity` within the event loop, as shown in the following example:

```
sysActivity : SystemActivity;
myRendezvous : Rendezvous = new();
myRendezvous.SetMode(mode = CM_CM_BLOCK);
event loop
  sysActivity =
    task.Part.OperatingSystem.ActivityManager.
      RegisterSystemActivity(systemId = fileId,
      activityType = SH_SA_FD_WRITE,
      userContext = pContext, rendezvous = myRendezvous);
  when myRendezvous.SystemActivityCompletion do
    -- Execute TOOL code
end event;
```

When the Activity Completes

When your application is notified about a completed system activity, either by retrieving information using the `GetSystemActivity` method or by the SystemActivityCompletion event, the task can proceed to do any other application specific tasks. For example, the application can perform UNIX I/O on the registered file descriptor or update data after signal or asynchronous trap (AST) notification.

NOTE When an AST or signal completes, the running process is no longer at interrupt or AST level on the operating system.

General Design Suggestions

In your application, you can start tasks specifically for handling system activities. This set of tasks can retrieve system activity information from the Rendezvous object, then take appropriate action based on the system activity.

Each invocation of the `Rendezvous.GetSystemActivity` method retrieves the first system activity from the queue of system activities that the Rendezvous object has received. Coordinating when different system activities are retrieved by different tasks from the same Rendezvous object can become a complex task if many tasks and system activities are involved.

We recommend the following guidelines to simplify the interaction among tasks and Rendezvous objects:

- Define one event loop for each Rendezvous object, so that tasks in only one event loop can retrieve system activities from a particular Rendezvous object.
- Define one Rendezvous object for each system activity for which the task registers. This approach prevents a task from retrieving a system activity of the wrong type.
- Define a means of dealing with the situation of a task not receiving an expected system activity. For example, your TOOL application should deal with the situation of receiving a Rendezvous.SystemActivityCompletion event, but not finding any system activities in the queue of the Rendezvous object.

Available Interfaces

You can take different approaches to waiting for system activities, depending on whether you use the iPlanet UDS classes, the C++ handle classes, or the C interfaces provided for the `ActivityManager` and `Rendezvous` classes.

iPlanet UDS applications iPlanet UDS applications can register and wait for notification as described for the `ActivityManager`, `Rendezvous`, and `SystemActivity` classes.

C++ client applications C++ client applications can use the C++ handle classes, as described in Chapter 14, "Accessing iPlanet UDS Using C++," to use the `ActivityManager`, `Rendezvous`, and `SystemActivity` classes. C++ client applications cannot receive iPlanet UDS events, so they must either poll or block to be notified about a system activity. If the C++ client application is single-threaded, then the Rendezvous object should be in polling mode.

C functions Because many iPlanet UDS application integration programs are written in C, iPlanet UDS also provides a C language interface so C programs can access the iPlanet UDS wait mechanism directly with better performance. C programs can use the C interfaces described for the `ActivityManager` and `Rendezvous` classes to register for system activities and then either poll to check or block to wait for notification that a system activity has occurred. The C interfaces work essentially the same as their counterparts in TOOL, except that identifiers are returned to represent the underlying objects.

If the C function is single-threaded, then the Rendezvous object should be in polling mode.

You need to include the header file for these functions, `sysact.h`, in your C code. This file is located in `FORTE_ROOT/install/inc/cmn`.

You must wrap these C functions, as described in *Integrating with External Systems*, so that the iPlanet UDS runtime system can link in these libraries and the C functions and the iPlanet UDS runtime system can talk to each other.

For syntax descriptions of the C interfaces, refer to the iPlanet UDS online Help.

Setting Up User-Defined Activities

You can create and post your own activities that can be used as callbacks to an iPlanet UDS application from a C, C++, or TOOL application. In particular, an external C client can notify iPlanet UDS about an external happening and pass data to iPlanet UDS using the `userContext` parameter of the `ActivityManager.PostSystemActivity` method (or its C and C++ equivalents, as described below.)

To set up and use a user-defined activity, you need to select a systemID for the activity. The systemID value is a completely arbitrary integer value.

➤ **To post a user-defined activity**

1. In the application that is waiting for the activity, register for the activity using the `ActivityManager.RegisterSystemActivity` method (qqsh_ASTRegisterSystemActivity or qqsh_RegisterSystemActivity function).
2. Pass the SystemActivity object returned in Step 1 to the application that will post the activity.

3. Post the user-defined activity using one of the following based on the language you are using:

   In TOOL, use the `ActivityManager.PostSystemActivity` method, described in the iPlanet UDS online Help.

   In C++, use the `qqhActivityManager.PostSystemActivity` method. See the iPlanet UDS online Help for information.

   In C, use the qqsh_PostSystemActivity function call, described in the iPlanet UDS online Help.

   The `notificationID` parameter is the SystemActivity object passed to the posting application in Step 2 above. The `systemID` parameter is the value selected by the user for this activity. You can choose whether to specify the `userContext` parameter, depending on the purpose of the activity.

   The `PostSystemActivity` method (or its C or C++ equivalent) notify the Rendezvous object associated with the registration that the user-defined activity has occurred.

Using the ExternalConnection Class

The purpose of the ExternalConnection class is to facilitate network communications between two points (peer to peer or client to server) when one of the points is an iPlanet UDS application. Using the ExternalConnection class, an iPlanet UDS application can communicate with an external process or program that is running locally or on another host. For example, an iPlanet UDS application can exchange data with a Web browser, Java, C, or BASIC program, telnet, HTTP, and so on.

Reference information for the methods of the `ExternalConnections` class is in the iPlanet UDS online Help.

You can use the ExternalConnection class to initiate *outbound* connections (using the Open method) or to listen for and accept *inbound* connections (using the `StartListening` method). Connections can occur over the following transport protocols:

- TCP (Sockets, TLI, Winsock)
- DECnet
- UNIX Domain Sockets

Advantages

The `ExternalConnection` class offers a high-performance, generic means of integrating with iPlanet UDS, by enabling connections to software programs on any hardware platform, using proprietary as well as standard Internet protocols. You can use wrapping C functions as an alternative to the `ExternalConnection` class, as described in the iPlanet UDS manual *Integrating with External Systems*. However, the `ExternalConnection` class offers several benefits compared to wrapping C functions:

The ExternalConnection class can improve performance. Whereas wrapper code typically uses synchronous processing, the `ExternalConnection` class supports fully asynchronous processing (non-blocking) in both clients and servers.

It is easier to use. The `ExternalConnection` class offers a standard, yet flexible, way to open connections. Wrapping C functions tend to be complex and unique to each situation.

It transparently handles network protocol details. The ExternalConnection class sets up non-blocking communication, coordinates polling, and asynchronous I/O completion. Additionally, it converts a number of protocol-specific error codes to a few standard iPlanet UDS exceptions.

Note that `ExternalConnection` is non-blocking when used in conjunction with iPlanet UDS's tasking model. The use of iPlanet UDS tasks is required to achieve multi-threaded operation with `ExternalConnection`.

This class can be used to include higher-level protocols such as SNMP, HTTP, FTP and other proprietary messaging systems in TOOL code. Examples of how you can use the `ExternalConnection` class include:

- on the client side, communications between client applications written in any language and an iPlanet UDS business service object
- on the server side, communications between an iPlanet UDS business service object and proprietary hardware or complex C processes
- on either the client or the server side, building an SNMP interface to iPlanet UDS applications to report to a proprietary system management utility

iPlanet UDS sample programs iPlanet UDS provides two sample iPlanet UDS programs (InboundExternalConnection and OutboundExternalConnection) to demonstrate the use of the `ExternalConnection` class and a sample C program, extcon.c, that connects with both iPlanet UDS programs. For information about these sample programs, see "InboundExternalConnection" on page 346 and "OutboundExternalConnection" on page 354.

Types of Connections

The `ExternalConnection` class supports network communications over multiple transport protocols. The following table shows the network transport protocols supported by the `ExternalConnection` class; note that one advantage of using `ExternalConnection` is that the protocol details are transparent on the iPlanet UDS side of the connection.

Type of Connection	From	To
TCP/IP Sockets (BSD or Winsock)	Windows NT	any reliable TCP/IP entity
	Digital Unix (Alpha OSF)	
	HP/UX	
	AIX	
	DG/UX	
TCP/IP TLI Endpoint (System V)	Dynix PTX	any reliable TCP/IP entity
	Solaris	
UnixDomainSockets (UDS)	Digital Unix	any other process, running on the same machine, that can read and write UDS
	HP/UX	
	AIX	
	DG/UX	
DECnet	Windows (Pathworks)	any DECnet entity
	VMS	

ExternalConnection objects are fully asynchronous in both preemptive and non-preemptive tasking environments. The task/thread blocking that supports asynchronous behavior is fully integrated into the iPlanet UDS task manager.

Basic Concepts

Some new terminology is introduced with the `ExternalConnection` class. Additionally, some terms can be ambiguous without context, since they may refer to either end of a network connection. We use the following terminology throughout this section and a number of the elements are shown in Figure 16-3 on page 320.

The *external endpoint* is the end of a network connection that is not running iPlanet UDS, but that must interact with an iPlanet UDS application. This endpoint can be a proprietary or standard software program or hardware component. The external program can be written in any language that supports an interface to the protocols shown in the table above, such as C or Basic.

The *iPlanet UDS endpoint* is the end of a connection where iPlanet UDS is running. An iPlanet UDS endpoint can be a partition containing a service object or a client partition.

Both endpoints can send data, receive data, or both. Similarly, either endpoint can initiate a connection. From the iPlanet UDS endpoint's perspective, a connection can be considered either inbound or outbound.

A *network connection* is maintained by the underlying network protocol such as TCP. A network connection supports data flowing in both directions simultaneously, such that both endpoints can be simultaneously reading and/or writing.

The *listener* is a task that is listening for incoming connections. A listener may accept connections on behalf of iPlanet UDS or on behalf of a program running external to iPlanet UDS.

An *iPlanet UDS listener* is a task currently listening for incoming connections. Typically the listener is started as a separate *listener task* dedicated to listening, although the listener can run in any iPlanet UDS task. The iPlanet UDS listener blocks while waiting for new incoming connections.

The *external listener* is a process at the external endpoint to which the iPlanet UDS application (for example, a business service object) sends a connection request.

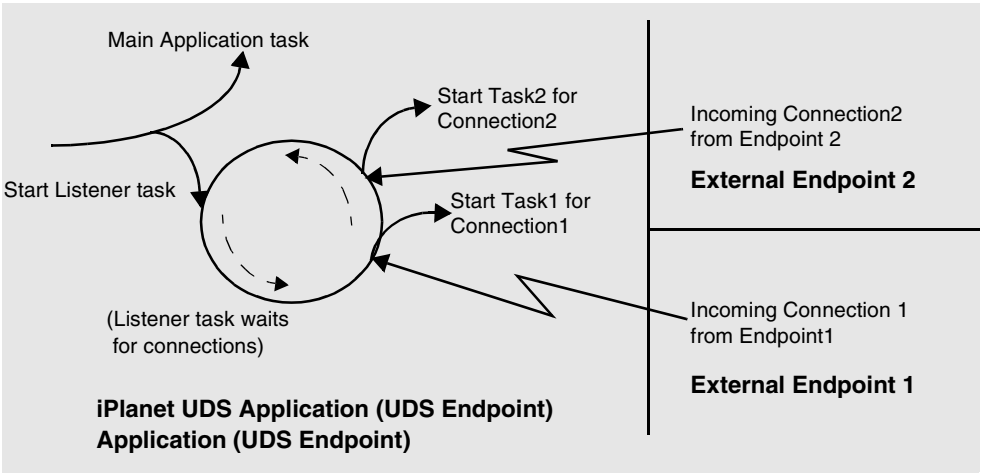
A *processing task* is an iPlanet UDS task started by the listener to set up the network connection and exchange data. When the listener starts the processing task it passes an ExternalConnection object for the new network connection. The final network connection is between the external endpoint and the processing task at its own port number.

Accepting Inbound Connections

An iPlanet UDS application might need to accept connections or requests that are initiated by external processes, such as C programs, management consoles, and so on. In your iPlanet UDS program, use the StartListening method (see the iPlanet UDS online Help) to start a listener task that can accept inbound connections.

In a typical iPlanet UDS application, a listener is created as a separate iPlanet UDS task, to avoid blocking the main task, as shown in Figure 16-3. The listener task blocks while it waits for new connection requests. When a connection request arrives, the listener task starts a new iPlanet UDS processing task for each new connection.

Figure 16-3 iPlanet UDS Listener Task Accepting Inbound Connections



An ExternalConnection object manages the underlying network connection. When using an ExternalConnection object, each read or write (or listen) method call blocks the task that issued it. Thus, you need multiple tasks whenever you want to have multiple operations outstanding.

The following code fragment shows a loop in which the listener starts listening and starts a task whenever it receives a new connection request:

```
newConn : ExternalConnection;
listener : ExternalConnection = new;
while TRUE do
  newConn = listener.StartListening(port);
  if (newConn != nil) then
    Start Task self.ProcessConnection(newConn);
  else
    exit;
  end if;
end while;
```

Project: InboundExternalConnection • **Class:** Connector • **Method:** Listen

The following example shows how to start a listener task:

```
Start Task ConnectSvr.Listen(port);
```

Project: InboundExternalConnection • **Class:** RunAll • **Method:** Runit

The following example shows how to subsequently start a processing task whenever a new connection arrives:

```
Start task self.ProcessConnection(newConn);
```

Project: InboundExternalConnection • **Class:** Connector • **Method:** Listen

Processing tasks In most cases it is preferable start a separate processing task to respond to each incoming connection. If you do not start separate processing tasks and the listener handles every incoming request, the listener will block while processing each request, severely impacting performance. By using processing tasks, the listener can accept incoming requests as quickly as they come in. For additional information on the `start task` TOOL statement and multitasking, see the *TOOL Reference Guide*.

In the interval between the time the listener accepts the connection and generates the new ExternalConnection object to pass to the processing task, the underlying network protocol buffers any data being sent, so data is not lost.

Because processing tasks are asynchronous, no task blocks waiting for another to complete.

The listener task On most platforms and protocols, a listener can accept as many connection requests as arrive, even "simultaneously." However, it is possible that more requests may arrive than can be serviced by a port number that has a limit. On many machines this limit is 5. If this situation occurs, the requestor should get an error and can retry. This problem can occur with some implementations of BSD sockets.

The listener will listen as long as the service object that started it is running, the listener task has not been closed, and the partition has not exited. The listener is closed if the partition exits or if the task that created the ExternalConnection object dies, or a Close is invoked on the listener. The listener task dies if Close is invoked on the ExternalConnection object.

This dependency may affect which service object you use to start the listener task. You can use one of the following approaches to start a listener task:

- You can use an existing business service object (the one that will either send or receive the data through external connection) to start the listener task.

  This approach is somewhat simpler to implement and is usually sufficient. It is also the approach taken in the sample program InboundExternalConnection.

- You can create a new service object specifically for the purpose of starting the listener task.

  If you need a listener to listen when the business service object cannot be guaranteed to be running, then you should either use a different service object, or create a new service object specifically to start the listener task. This allows you to explicitly control when that service object (and hence, the listener) starts and stops.

Data transfer After the two endpoints establish a connection, they can exchange data. At the iPlanet UDS endpoint, each processing task uses an ExternalConnection object for one connection, and the iPlanet UDS application writes and reads from the object as required by the business needs. The ExternalConnection object sends data to and receives data from the network using a MemoryStream object as an intermediate buffer. For example, if the iPlanet UDS endpoint is sending data to an external endpoint, it first writes data to the MemoryStream buffer and then uses the Write method of ExternalConnection to send the data from the buffer over the network to the endpoint.

Using the data at the endpoints At the external endpoint, a program receives the data that is sent by the iPlanet UDS endpoint, or sends data to the iPlanet UDS endpoint. Before iPlanet UDS sends object data to an external endpoint, the iPlanet UDS program must convert object data into scalar data, so the data can be used by the endpoint program. Conversely, when an external endpoint sends data to iPlanet UDS, the scalar data may need to be converted into objects. Both endpoints can then manipulate or display the data as desired.

Closing the connection When the connection is no longer required and all information has been passed, the connection should be closed. Use the `Close` method (see the iPlanet UDS online Help) to close an ExternalConnection object. Although iPlanet UDS automatically invokes `Close` under some circumstances, it is good practice to explicitly invoke `Close`.

Making Outbound Connections

You might require your iPlanet UDS application to initiate calls to an external system to pass or request data. To initiate a connection from an iPlanet UDS application, use the `Open` method (see the iPlanet UDS online Help).

For each connection you must identify a host and a network endpoint:

- To identify a host, specify a name in the current domain, a fully qualified domain name, or an IP address.
- To identify a network endpoint, specify a port number, path name, or DECnet object name.

Data transfer during an outbound connection is the same as for an inbound connection.

Using MemoryStream Buffers

To read data from and write data to an ExternalConnection object, you use the Read and Write methods. Both Read and Write use MemoryStream objects as data buffers, with a parameter called readLength or writeLength.

NOTE For more efficient code, use the `UseData` method on MemoryStream to preallocate contiguous buffers in the desired size for the data to be transferred. Set the size to the largest typical size of data to be transferred. (Note that data buffers may be temporarily broken up during transmission over the network.) If you do not use the UseData method, buffers are allocated as necessary, but performance may be degraded

The following code fragment shows the use of UseData. Note that the markers <EOW> and <EOS> are specifically used by the iPlanet UDS examples to indicate the end of a word or string; these markers are not automatically used or supported by the `ExternalConnection` class.

```
feedData.SetValue('Tire<EOW1>65psi<EOW2>Inflated<EOW3><EOS>');
length = feedData.ActualSize;
-- If you have a large amount of data, it's more efficient to
-- use UseData() than any of the write methods on MemoryStream.
buf.UseData(data = (pointer to char)(feedData.Value),
       length = length);
conn.Write(buf, length);
```

Project: OutboundExternalConnection • **Class:** Connector • **Method:** Connect

Other recommendations include:

- Open the buffer with the appropriate access mode (read, write, or read/write).
- Reuse data buffers as much as possible to avoid allocation and memory management overhead.
- Have a read posted on each I/O connection when you are expecting data to arrive, to increase throughput and response.

The following example invokes the Read method within the `ProcessConnection` method immediately after opening the MemoryStream buffer, as follows (ellipses denote missing comments, not actual code lines):

```
buf     : MemoryStream = new;
length  : integer;
...
readComplete : boolean;
...
buf.Open(SP_AM_READ_WRITE);
while TRUE do
  length = MAXLEN;
    . . .
    while not readComplete do
    . . .
    newConn.Read(buf, length);
    buf.ReadText(target = tempTD, length = length);
    end while;
end while;
```

Project: InboundExternalConnection • **Class:** Connector
• **Method:** ProcessConnection

For more information on using MemoryStream, refer to the section on MemoryStream in the Framework Library online Help.

Data Sharing Issues

When an iPlanet UDS application and an external program share or exchange data, consider the following issues:

Map objects to scalar data. You must provide mappings that convert object data (such as TextData or IntegerNullable) to scalar data (such as string or int). Data that is passed from an iPlanet UDS application must be converted before transmission into data types that can be handled by the external program.

Use same high-level protocol. The programs at both endpoints must write and read data using the same high-level protocol (for example, HTTP, FTP, Telnet, and so on). While the lower-level transport protocol details are made transparent by using the ExternalConnection class, the iPlanet UDS application and the external program will probably use some higher-level protocol to exchange and interpret data.

For example, if you are sending iPlanet UDS objects to be displayed on a Web server using the HTTP protocol, then you must embed the necessary HTTP information when sending the data, so that the data can be interpreted properly by HTTP at the endpoint.

Map data to iPlanet UDS objects. If data is being passed *to* the iPlanet UDS application, you may need to define one or more new classes and attributes to which the data will be mapped. However, you need not map all data to objects; for example, you might also pass data values to be used for local variables.

Take special care with binary data. If you are passing binary data, remember to allow for byte-swapping when passing data between platforms that use different byte-ordering format for binary data. For example, the VMS, NT, Intel, and Sequent platforms use one byte-ordering format, that differs from that used by Sun, HP, Mac, and AIX. Also, when passing binary data, make sure that the sizes of the datatypes are compatible; for example, on some platforms an int datatype is 4 bytes, while on other platforms it is 2 bytes.

Scaling Issues

An application might have multiple listeners waiting to accept connections. The following scenarios are valid possibilities and are described in more detail below:

- The main partition task can start listeners on different port numbers. This might be useful to allow incoming connections to access different service objects. Each service object would have a corresponding listener at a unique port number, for example.
- Different tasks within the same application can start parallel listener tasks, to accept connections that will be serviced by different service objects. Or, one task can start several listener tasks that all call the same service object.
- The main partition task can start multiple listener tasks, one for each underlying network protocol. For example, it could start two listener tasks: one for TCP sockets and one for DECnet. In this case, incoming connections could access the same service object but use different underlying transport protocols.

Using multiple listeners You can start multiple listeners for a variety of reasons. If external connections may require access to multiple service objects, you can set up listeners at different locations (ports or DECnet objects) for each service object. One advantage of this is that the connections can be processed in parallel.

Using multiple transport protocols concurrently You can design your application to communicate over multiple transport protocols (for example, DECnet and TCP sockets) simultaneously. To use multiple protocols you must define one listener for each protocol (for inbound connections) or open individual connections for each protocol (for outbound connections).

Using Multiple Tasks for a Single Connection

We recommend that you use only one iPlanet UDS task to read or write on an ExternalConnection object. This has the advantage of being far simpler to code and easier to manage. However, it is possible to start multiple tasks to read or write to the same ExternalConnection object.

For example, you might prefer to start multiple tasks to handle a connection, to quickly transfer large or complex data. For example, an external endpoint might require (or send) a continuous feed of data and images. In such situations, you may decide to start separate tasks to read or write portions of the data.

Although using multiple tasks can be desirable for performance reasons, it requires more complex code than using only one buffer, or than always processing buffers in the same order (as from a single task). If you use multiple tasks to process multiple buffers, consider the following issues:

Synchronize buffer access. If the tasks are asynchronous, the buffers are processed in parallel and you must synchronize the processing. Your code must synchronize and lock the data from the multiple buffers. Since you cannot predict in what order the tasks will complete, your code must be sure to construct (or interpret) the buffers correctly, no matter what order the buffers are sent (or received). (If the tasks are not asynchronous, then, while the coding of the multiple tasks is simpler, you lose any potential benefit due to using multiple tasks.)

Minimize object contention. The primary reason to use multiple tasks for one connection is to improve performance. However, by using multiple tasks for one ExternalConnection object you increase contention on that object; depending upon the situation and the code, you may or may not realize a performance improvement by using multiple tasks instead of a single task.

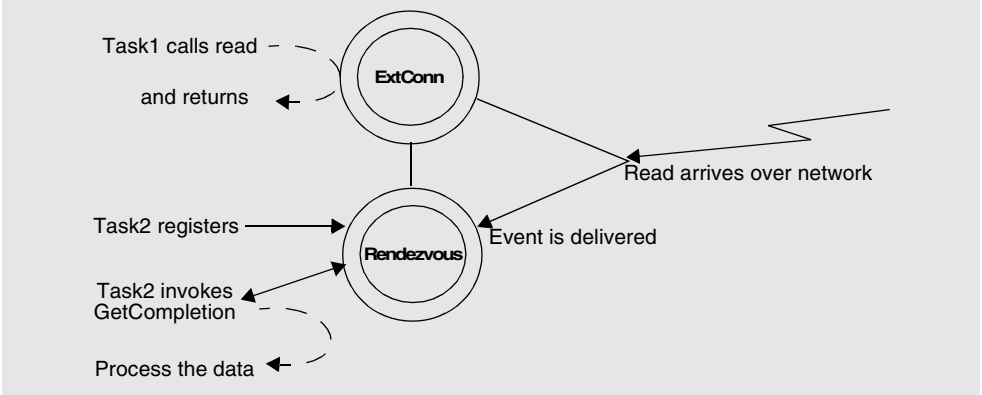
Using Task-Level Asynchronous Reads

Because ExternalConnection I/O is asynchronous to the partition, tasks not involved in the I/O are free to perform other work and multiple requests can be processed in parallel. This model is sufficient for many types of applications including servicing of HTTP requests. However, this model does not scale well for a subset of applications, in particular, those that have long-lived sessions.

Standard ExternalConnection I/O In the standard ExternalConnection I/O model, a task blocks while waiting for a read to complete. If there is significant think time in the client, and connections or sessions last longer than an I/O, then these blocked tasks can consume a large amount of memory on the server. (Note that HTTP connections are not subject to this problem, since the connection only lasts one I/O.)

Task-level ExternalConnection I/O An alternative model for applications that have long-lived sessions uses task-level asynchronous reads with ExternalConnection objects. In this model, a task that does a read never blocks waiting for the I/O to complete. Once the I/O has completed it is delivered to an object known as a *rendezvous*. When using asynchronous reads with a rendezvous, you can have a pool of tasks that wait on the Rendezvous object for reads to complete asynchronously. In this way, tens of tasks can service thousands of session connections reducing memory overhead.

Figure 16-4 Using Asynchronous Reads with a Rendezvous Object



You can create as many Rendezvous objects as you require for an application; using multiple Rendezvous objects can be a convenient way to arrange work in an application.

To enable asynchronous reads, first create a Rendezvous object and then call the `SetIORendezvous` method on the ExternalConnection object. At that point a normal read will return immediately with a length of 0. For example:

```
-- Mark the I/O async
rendezvous : Rendezvous = new;
Connection.SetIORendezvous(self.Rendezvous);
-- do the I/O
buf.Seek(SP_RP_START, 0);
length = 50;
Connection.Read(buf, length);
-- length is 0 now.
-- perform other work ....
```

The same task or another “completion” task can then wait for the I/O to complete:

```
event loop
        when rendezvous.IOCompletion do
        length = Rendezvous.GetCompletion(retConn, retStrm);
        if length = -1
        then
                -- No completion found.
                exit;
        end;
        if length = 0
        then
                -- close seen.
                return;
        end;
        task.Part.Stdout.Put(' read got back ');
        retStr.Replace(length);
        task.Part.Stdout.Put(retStr.AsCharPtr());
        task.Part.Stdout.Put(' bytes\n');
end event;
```

The completion task can either poll or wait for the I/O.

Waiting for I/O To wait for the I/O, the task enters an event loop registered for a Rendezvous.IOCompletion event.

Polling for I/O To poll for a read completion, the task periodically calls the method Rendezvous.GetCompletion.

To turn off asynchronous reads, invoke the `ClearIORendezvous` method.

There may be only one read posted to a connection at a time; the first read is accepted and subsequent reads are ignored.

Error Handling

A number of situations are automatically handled by the `ExternalConnection` class. For example, if a partition in which the listener task is running is shut down, then iPlanet UDS automatically closes any outstanding connections and reallocates the resources.

Network connections may experience problems due to the underlying transport protocol. RemoteAccessException errors are raised when a connection cannot be established, a current connection fails, or when network addresses cannot be resolved. The individual method descriptions list the exceptions that each method may raise.

Under some circumstances the iPlanet UDS side of a connection may close before the external connection has received all of the data. This may occur, for example, on a fast machine that closes a socket after having sent the data but before the data traverses the network. The Close method includes a slight delay to catch network problems at this point. If desired, you can also add a pause, using the Timer class, before the socket closes. You may want to experiment some to determine the optimal value for the pause; values less than 50 milliseconds are effectively a task-yield, since the operating system does not have the granularity to measure times that small.

The following code contains error handling code that checks for three types of exceptions and explicitly closes the connection in the event of a RemoteAccessException:

```
exception
   when ue : UsageException do
      task.Part.LogMgr.Put(
        'UsageException caught in ProcessConnection\n');
   when sre : SystemResourceException do
      task.Part.LogMgr.Put(
        'SystemResourceException caught in ProcessConnection\n');
   when rae : RemoteAccessException do
-- This exception means the connection was lost. We
-- explicitly close our connection to avoid any timing issues
-- in the cleanup. If you don't explicitly Close the
-- connection, it will get cleaned up at the end of the
-- next garbage collection.
      task.Part.LogMgr.Put(
        'RemoteAccessException caught in ProcessConnection\n');
      task.ErrorMgr.Remove(1);
      newConn.Close();
-- Post Event so main task knows to shut down.
      post self.ConnectionClosed;
```

Project: InboundExternalConnection • **Class:** Connector
• **Method:** ProcessConnection

Diagnostics for ExternalConnection

When debugging and tracing multiple connections, you can use the `SetName` method to assign a unique name to each connection, and the method GetName at appropriate intervals in your code to verify which connection is current.

You can also use the CommMgr agent in the Environment Console to track the reads, writes, bytes sent and received, and opens and closes. The CommMgr agent manages the communications service for an active partition. For example, the CommMgr agent has instruments that represent reads, writes, and bytes sent or read (using the corresponding instruments Recvs, Sends, BytesSent, and BytesReceived).

For more information on the CommMgr agent and its instruments, see the *Escript and System Agent Reference Guide*.

Appendix A

iPlanet UDS Example Applications

This appendix contains the following information for example applications used in this book:

- a brief description of each example
- how to install the examples
- requirements for running examples
- instructions for running the examples

You can run an example application and examine it in the various iPlanet UDS Workshops to see how it is implemented. You can also modify the examples to use them as starting points for your own applications.

Overview of iPlanet UDS Example Applications

This section provides an overview of the iPlanet UDS example applications, organized by general topic. The following tables list the example applications under the particular part of the iPlanet UDS system they demonstrate.

For the complete description of an individual application, see "Application Descriptions" on page 336, which lists the applications in alphabetical order.

ActiveX Examples

Table A-1 .pex files located in FORTE_ROOT/install/examples/extsys/ole/client

Example	Description
ActiveXDemo	Uses the iPlanet UDS FourDir ActiveX control in an iPlanet UDS window.

C

Table A-2 .oex files located in FORTE_ROOT/install/examples/extsys/c/

Example	Description
AllCType	Maps TOOL C data types to variables in C functions. All C data types are covered.
DMathTm	Shows how to integrate TOOL code with C functions in a distributed application.
MathTime	Shows how to integrate TOOL code with C functions.
XRefTime	Shows how to free external resources based on TOOL memory reclamation.

C++

Table A-3 .pex files located in FORTE_ROOT/install/examples/extsys/cpp/server

Example	Description
CppBanking	Shows how to provide a C++ API to external C++ client applications.

DDE Examples

Table A-4 .pex files located in FORTE_ROOT/install/examples/extsys/dde/

Example	Description
DDEClient	Illustrates the use of the `DDE Conversation` class; iPlanet UDS is the client.
DDEServer	Lets an iPlanet UDS application act as Microsoft Windows DDE server application.

ExternalConnection

Table A-5 .pex files located in FORTE_ROOT/install/examples/extcon

Example	Description
InboundExternalConnection	Illustrates how to use the `ExternalConnection` class to listen for a connection.
OutboundExternalConnection	Illustrates how to use the `ExternalConnection` class to initiate a connection.

OLE Examples

Table A-6 .pex files located in FORTE_ROOT/install/examples/extsys/ole/server and FORTE_ROOT/install/examples/extsys/ole/clientr

Example	Description
OLEBankEV	Creates an environment-visible service object and an OLE client that can interact with the service object.
OLEBankUV	Creates an user-visible service object and an OLE client that can interact with the service object.
OLESample	Illustrates the use of OLEField, `Olegen`, and iPlanet UDS's OLE Automation.

XML Server Example

Table A-7 .pex file located in FORTE_ROOT/install/examples/xmlsvr

Example	Description
BankServices	Exports basic bank services as an XML server. This project is based on the bank services project used in the tutorial for *Getting Started With iPlanet UDS* manual.

The BankServices server provided with BankServices.pex can be accessed from the Java clients in the following package:

```
com/forte/xmlsvr/examples
```

Table A-8 .java files located in the package at FORTE_ROOT/install/examples/xmlsvr

Example	Description
BankClient.java	Command line Java client with hard-coded calls to the XML server created in the BankServices project. This Java client illustrates basic usage.
Welcome.java DisplayAccountDialog.java NewAccountDialog.java	Swing Java client classes, demonstrating access to an XML server from an interactive client.

Application Descriptions

This section lists the example applications in alphabetical order. Each example has five sections describing it.

The **Description** section defines the purpose of the example, what problem it solves, and what TOOL features and iPlanet UDS classes it illustrates.

The **Pex Files** section gives you the subdirectory and file names of the exported projects. The examples are in subdirectories under the `FORTE_ROOT/install/examples` directory. You can import example applications individually if you wish. When multiple .pex files are listed, there are supplier projects in addition to the main project. You will need to import all the files listed to run the application. Import them in the order given so that dependencies will be satisfied.

The **Mode** section indicates whether the application can be run in either standalone or distributed mode, or whether it must be run in distributed mode.

The **Special Requirements** section identifies whether you need a database connection, an external file, or any other special setup.

Finally, the **To Use** section tells you how to step through the application's functions.

See the *iPlanet UDS System Management Guide* if you need directions to set up an iPlanet UDS server. See *Accessing Databases* if you need information on how to make a connection to a database. The database examples run against either Sybase or Oracle.

ActiveXDemo

Description ActiveXDemo shows how to use ActiveXField widgets to display and interact with ActiveX controls. To use this example, you need the following files:

- `actxsamp.pex`, which contains a project that uses ActiveX fields to interact with the ActiveX controls
- FourDir ActiveX control file, which is one of the following, depending on the platform:

Platform	File name for FourDir ActiveX control file
Windows 95	`fdir32.ocx`
Windows NT	`fdir32.ocx`
Windows NT on Alpha	`fdirant.ocx`

Pex File `extsys\ole\client\actxsamp.pex`.

Mode Standalone.

Special Requirements Windows 95 or Windows NT.

➤ **To use ActiveXDemo**

1. Copy the `.ocx` file for the FourDir ActiveX control to another location on your system. If you wish, you can copy `olegen.exe` from `FORTE_ROOT\install\bin` directory to the same location.
2. Register the FourDir ActiveX control in the Windows registry. The following table shows how to register the ActiveX control for each platform:

Platform	Command to Register the Control
Windows 95	`regsvr32 fdir32.ocx`
Windows NT	`regsvr32 fdir32.ocx`
Windows NT on Alpha	`regsvr32 fdirant.ocx`

   The `regsvr.exe` and `regsvr32.exe` are distributed with iPlanet UDS in the `FORTE_ROOT\install\bin` directory.

   For example, in Windows 95 or Windows NT, use the following command on the command prompt in the directory that contains the .ocx file for the control:

   ```
   regsvr32 fdir32.ocx
   ```

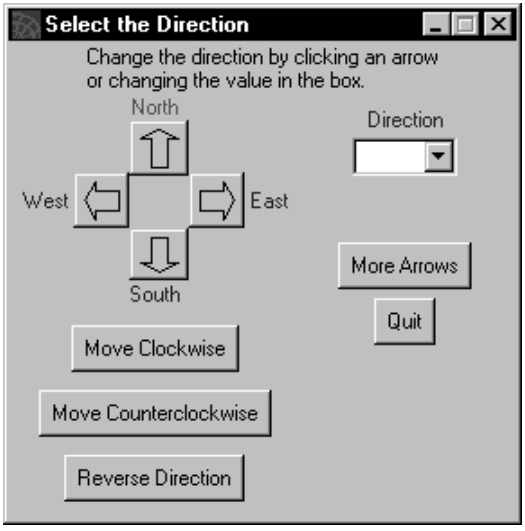
3. Use the `Olegen` utility to generate a pex file based on the .ocx file using a command like the following:

   ```
   olegen -it fdir32.ocx -of fdir32.pex -ai
   ```

   This command generates a .pex file called `fdir32.pex`.
4. Import the generated .pex file into your repository.
5. Import the `actxsamp.pex` file into your repository.
6. Test run the ActiveXDemo application.

The window in the ActiveXDemo application is shown in the following figure:

Figure A-1 ActiveXDemo window



The four arrows in this window belong to the FourDir ActiveX control in an ActiveX field. The direction indicated by the selected arrow is reflected in the Direction droplist, and vice-versa. Another ActiveX field is defined in the lower right hand corner, but invisible.

The functions provided by the buttons are described below:

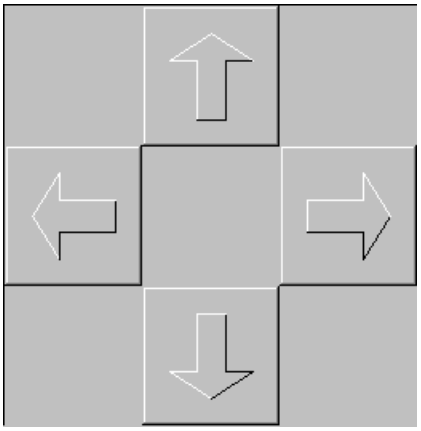
Button	Description
More Arrows	Sets the state of the invisible ActiveX field to update, then creates a new instance of the fdir class (the ActiveX interface class for the FourDir ActiveX control) and inserts it into the ActiveX field. When you click one of the arrows in this control, the selected arrow moves clockwise until it returns to the arrow you selected.
Move Clockwise	Makes the selected arrow the next one in a clockwise direction.
Move Counterclockwise	Makes the selected arrow the next one in a clockwise direction.
Quit	Shuts down this application.

Button	Description
Remove Arrows	Removes the FourDir ActiveX control added by the More Arrows button. When the More Arrows button is clicked, this button replaces it.
Reverse Direction	Makes the selected arrow the one pointing in the opposite direction from the arrow currently selected.

FourDir ActiveX Control

The FourDir ActiveX control is a sample ActiveX control provided by iPlanet UDS to demonstrate how you can use ActiveX controls in your iPlanet UDS applications.

Figure A-2 The FourDir ActiveX Control



The FourDir ActiveX control is provided as an .ocx file, as described in "ActiveXDemo" on page 337. This section also describes how to install the FourDir ActiveX control.

The following table outlines the methods, properties, and events defined for the FourDir ActiveX control:

Element	Name	Description
Property	Value	String. Sets the initial direction for the selected arrow. Valid values are N (north), S (south), E (east), or W (west). By default, the initial value is N.
Method	MoveClockwise	No arguments and no return value. Changes the selected arrow to the next arrow in the clockwise direction.
	MoveCounterClockwise	No arguments and no return value. Changes the selected arrow to the next arrow in the counterclockwise direction.
	MoveOpposite	No arguments and no return value. Changes the selected arrow to the arrow in the opposite direction.
Event	Click	Posted when someone clicks an arrow in the control.

AllCType

Description AllCType shows how to map TOOL C data types to variables in C functions. The "mapping methods" in this example are the methods defined in a TOOL C project which enable TOOL methods to access C functions. The methods and functions in this example perform extremely simple operations. Their purpose is to show how to define input, output, and input output parameters, and return values in the mapping methods, and how to call those methods from TOOL, and how to de-reference the parameters in the C functions. Also see the MathTime and DMathTm examples for a simple, practical example of how to use mapping methods. DMathTm is the distributed version of MathTime.

Pex Files `extsys/c/allctype.pex`.

Mode Distributed only.

Special Requirements Access to a C++ Compiler, creation of a working directory, autocompile must be turned on.

➤ **To use AllCType**

1. Create a working directory where you have read and write permission. Copy the following three files from `$FORTE_ROOT/install/examples/extsys/c` to your working directory: `allctype.pex`, `allctype.fsc`, `allctype.c`. Set the environment variable FORTE_EP_WRKDIR to your working directory.
2. Compile the file `allctype.c` into an object file called `allctype.o`. Under the directory `$FORTE_ROOT/tmp`, create the directory '`examples`', if it isn't there already. Copy the file `allctype.o` to the `$FORTE_ROOT/tmp/examples` directory.
3. Before completing this step, make sure autocompile is available on your system. If autocompile is not set up, ask your System Administrator to set it up for you. Now run Fscript and enter the following commands:

```
UsePortable
SetPath %{FORTE_EP_WRKDIR}
Include allctype.fsc
```

4. The `allctype.fsc` script will import, distribute, compile, install, and run the AllCType example. You may want to examine this script to understand the steps involved in linking TOOL code with external C routines.

CPPBanking

Description CPPBanking shows how to create an iPlanet UDS service object for which you can generate a C++ API. This example also shows how to generate a C++ API and how to write a C++ client that uses the API.

- `cppbank.pex` contains the TOOL project CPPBanking, which contains a simple class and starting method that references the BankServer service object in the BankServices project.
- `cppbancl.cpp` is the C++ client application that uses the generated C++ API to access an iPlanet UDS client partition.

BankServer is a simple bank account service object. That lets clients query and update bank accounts.

Pex Files `frame/banksvc.pex`, `extsys/cpp/server/cppbank.pex`.

Mode Distributed only.

Special Requirements Have access to a C++ compiler, set up auto-compile, set up your environment and C++ compiler and linker, as described in "Setting up Your System and Compiler to Use the C++ API" on page 270.

➤ **To use CPPBanking**

1. Import the `banksvc.pex` (BankServices project) and `cppbank.pex` (CPPBanking project) files into your repository.
2. Partition the CPPBanking project.
3. Open the properties dialog for the client partition for the CPPBanking application.
4. Mark the Compile and Generate C++ API toggles and click the OK button.
5. Make a distribution for this application using auto-compile and auto-install.
6. Compile the `cppbancl.cpp` file using the compiler and linking options described in "Compiling the C++ Client Application" on page 283.
7. Start the executable created by compiling and linking `cppbankcl.cpp`.

DDEClient

Description DDEClient uses the DDEConversation class, which lets an iPlanet UDS application access a Microsoft Windows Dynamic Data Exchange (DDE) server application on a PC/Windows platform. It allows you to establish a connection with Excel and move data to an Excel spreadsheet.

Pex Files `extsys/dde/ddecli.pex`.

Mode Standalone or Distributed.

Special Requirements PC client running Excel, access to `extsys/dde/ddecli.xls`.

➤ **To use DDEClient**

1. Before trying to run this application, check the location of your Excel executable.

   If it is not in C:\EXCEL, edit the Display method in the DDEClientWindow class to point to the right directory. Enter the full path name to your Excel spreadsheet and click on the Connect button. If Excel is not running, it will be started and Already Running will be checked.

2. Place the windows so that both the DDEclient and Excel are visible. You can retrieve data from a particular cell of the spreadsheet by specifying the cell name and clicking the Get button. Similarly, you can place data by entering the Cell Value and clicking the Set button.

3. You can also change data in the Excel spreadsheet, click on the HotLink and WarmLink buttons, and note the status line at the bottom of the application. A hot link changes the data in the client display, while a warm link only notifies you of a change.

DDEServer

Description DDEServer uses the DDEServer and DDEClient classes, which let an iPlanet UDS application act as a Microsoft Windows Dynamic Data Exchange (DDE) server application on a PC/Windows platform. It is a simple utility for servicing a DDE client application.

Pex Files `extsys/dde/ddeserv.pex`.

Mode Standalone or Distributed.

Special Requirements PC client running Excel, access to extsys/dde/ddeserv1.xls and ddeserv2.xls.

➤ **To use DDEServer**

1. Start the application, then bring up Excel and open the example Excel files: `ddeserv1.xls` and `ddeserv2.xls`.

2. Select the StartTimer button. You should see the changing numbers in the iPlanet UDS server reflected in your example spreadsheets.

3. You can also use the appropriate menu items in Excel to retrieve data from the DDEserver application and place in the Excel spreadsheet, or to export data from the spreadsheet and place it in the DDEserver application.

DMathTm

Description DMathTm is the distributed version of MathTime. Both DMathTm and MathTime are examples of a TOOL C project, along with a TOOL project that calls the TOOL C project. They are both useful for seeing how to integrate TOOL code with C functions. DMathTm shows how to use a service object to restrict access to the C project. This is a realistic approach to accessing C functions in a distributed environment, since pointers cannot be passed across partitions. The example program allctype is a reference for how to define and call TOOL C methods with parameters of all C data types at assorted levels of indirection.

Pex Files `extsys/c/dmathtm.pex`.

Mode Distributed only.

Special Requirements Access to standard C Runtime Libraries and a C++ Compiler, creation of a working directory, autocompile must be turned on.

➤ **To use DMathTm**

1. Create a working directory where you have read and write permission. Copy the following three files from `$FORTE_ROOT/install/examples/extsys/c` to your working directory: `dmathtm.pex`, `dmathtm.fsc`, `dmathtm.c`. Set the environment variable FORTE_EP_WRKDIR to your working directory.
2. The file `dmathtm.pex` contains the C project DistMathAndTimeProject and the TOOL project TestDistMathAndTimeProject. They assume you have access to the standard C runtime libraries. Make sure you know where these are located and what they are called on your system.
3. Edit your copy of `dmathtm.pex` so that its ExternalSharedLibs extended property points to the standard C shared library. Search the file for the string '`/usr/shlib/libc`'. Change this string to the correct path and library name for your system.
4. Compile the file `dmathtm.c` into an object file called dmathtm.o. Under the directory `$FORTE_ROOT/tmp`, create the directory '`examples`', if it isn't there already. Copy the file `dmathtm.o` to the `$FORTE_ROOT/tmp/examples` directory.

5. Before completing this step, make sure autocompile is available on your system. If autocompile is not set up, ask your System Administrator to set it up for you. Now run Fscript and enter the following commands:

```
fscript> UsePortable
fscript> SetPath %{FORTE_EP_WRKDIR}
fscript> Include dmathtm.fsc
```

The `dmathtm.fsc` script will import, distribute, compile, install, and run the DMathTm example. You may want to examine this script to understand the steps involved in linking TOOL code with external C routines.

InboundExternalConnection

Description InboundExternalConnection illustrates how to use the ExternalConnection class to listen for a connection. The iPlanet UDS program waits for a new connection, then starts a task to handle each new connection. The C program extcon will initiate the connection this example is waiting for. Once the connection is established, data is read and written. For the read, the iPlanet UDS program checks for an end of string marker to make sure all the data is received.

Pex Files `inbound.pex`.

Mode Distributed.

Special Requirements C compiler; C portion of this example will run on NT and Unix platforms; it will not run on Mac, Windows, or VMS.

➤ **To use InboundExternalConnection**

1. Decide which platform you want to run the C program on, and which platform you want to run the iPlanet UDS program on. Compile the C program extcon.c into the executable extcon on the desired platform.

   On most Unix systems, simply use the following command:

   ```
   cc extcon.c -o extcon
   ```

This will work on the following platforms:

- AlphaOSF
- RS6000
- Solaris
- Data General

On Sequent, use the following command:

```
cc extcon.c -o extcon -lsocket -linet -lnsl
```

On HP, use the following command:

```
cc +Z extcon.c -o extcon
```

On NT, if you use Visual C to compile extcon.c, make sure to include `wsock32.lib` with your standard Object/Library modules. Also, make sure the application is defined as a console application, not a windows application.

2. Both the iPlanet UDS program and the C program will use a default port number for the listener, unless you supply it as an environment variable. The default port number is 6867. If you need to use another port number, set the environment variable FORTE_EP_REG_PORT_1 to the desired port number in both the environment where you will run the iPlanet UDS program and the environment where you will run the C program.

3. If you want to establish an external connection between the iPlanet UDS program and the C program running on the same Unix machine, you do not need to set an environment variable for the node name. If you want to connect between different machines, or if you want to make the connection on the same NT machine, you will need to set an environment variable. Set the environment variable FORTE_EP_NODENAME_1 in the environment where you are running the C program. Set it to the name of the machine where the iPlanet UDS program is running.

4. Use the file `inbound.scr` to supply the necessary commands to fscript. `Inbound.scr` will import the pex file inbound.pex, find the project, run it, and remove the project after the run is complete. `Inbound.pex` must be in the same directory as `inbound.scr`. Use fscript's -i flag to input inbound.scr to fscript:

   ```
   fscript -i inbound.scr
   ```

   Wait for fscript to import the project, load it, partition the service object, and return the client partition. The iPlanet UDS program is now waiting to accept an inbound connection.

5. On the machine where you compiled extcon, run it with the m command line option:

   ```
   extcon m
   ```

   When you use the m option, extcon attempts to make a connection.

6. Observe the output of both processes. On the iPlanet UDS side, you should see the following results:

```
Waiting to connect on port 6867
Waiting to connect on port 6867
Inbound Connection: server read got back 34 bytes
Lab<EOW1>50<EOW2>Stable<EOW3><EOS>
Inbound Connection: server wrote 35 bytes
Inbound Connection: server read got back 46 bytes
Storage Shed<EOW1>90<EOW2>Emergency<EOW3><EOS>
Inbound Connection: server wrote 35 bytes
Inbound Connection: server read got back 38 bytes
Vat<EOW1>200<EOW2>Red Alert<EOW3><EOS>
Inbound Connection: server wrote 35 bytes
Inbound Connection: RemoteAccessException caught in
ProcessConnection
Connection closed. All done.
```

From the C program, you should see the following lines:

```
Attempting to initiate connection on port 6867.
Attempting to initiate connection on the current machine.
Thank you for the information.
Thank you for the information.
Thank you for the information.
```

MathTime

Description MathTime is an example of a TOOL C project, along with a TOOL project that calls the TOOL C project. It is useful for seeing how to integrate TOOL code with C functions. The example program DMathTm is the distributed version of MathTime. The example program AllCType is a reference for how to define and call TOOL C methods with parameters of all C data types at assorted levels of indirection.

Pex Files `extsys/c/mathtime.pex`.

Mode Distributed only.

Special Requirements Access to standard C Runtime Libraries and a C++ Compiler, creation of a working directory, autocompile must be turned on.

➤ **To use MathTime**

1. Create a working directory where you have read and write permission. Copy the following three files from `$FORTE_ROOT/install/examples/extsys/c` to your working directory: `mathtime.pex`, `mathtime.fsc`, `mathtime.c`. Set the environment variable FORTE_EP_WRKDIR to your working directory.
2. The file `mathtime.pex` contains the C project MathAndTimeProject and the TOOL project TestMathAndTimeProject. They assume you have access to the standard C runtime libraries. Make sure you know where these are located and what they are called on your system.
3. Edit your copy of `mathtime.pex` so that its ExternalSharedLibs extended property points to the standard C shared library. Search the file for the string '`/usr/shlib/libc`'. Change this string to the correct path and library name for your system.
4. Compile the file `mathtime.c` into an object file called mathtime.o. Under the directory `$FORTE_ROOT/tmp`, create the directory '`examples`', if it isn't there already. Copy the file `mathtime.o` to the `$FORTE_ROOT/tmp/examples` directory.

5. Before completing this step, make sure autocompile is available on your system. If autocompile is not set up, ask your System Administrator to set it up for you. Now run Fscript and enter the following commands:

```
fscript> UsePortable
fscript> SetPath %{FORTE_EP_WRKDIR}
fscript> Include mathtime.fsc
```

6. The `mathtime.fsc` script will import, distribute, compile, install, and run the MathTime example. You may want to examine this script to understand the steps involved in linking TOOL code with external C routines.

OLEBankEV

Description OLEBankEV shows how to create an environment-visible service object that provides wrapper methods to the methods on other iPlanet UDS service objects. This example then provides a Visual Basic client that shows how an OLE client application could interact with the environment-visible service object to use the services provided by the BankServices.BankServer service object.

- olebanev.pex contains the OLEBankEV project, which defines a class named BankServiceOLEInterface, which defines wrapper methods that in turn invoke methods on the BankServices.BankServer service object. This project also defines an environment-visible service object called BankServerOLE.
- All the other files are part of the Visual Basic OLE client.

Pex Files `frame/banksvc.pex`, `extsys/ole/server/olebanev.pex`.

Mode Distributed only.

Special Requirements This example runs only on a Windows NT server node. You need to have Microsoft Visual Basic installed on the Windows NT server node and a C++ compiler. You should have autocompile set up.

The Visual Basic clients were written to use Visual Basic Version 4.0. If you are using later versions of Visual Basic, you might need to upgrade the provided Visual Basic components, adjust the following instructions.

➤ **To use OLEBankEV**

1. Import the .pex files, listed above.
2. Configure the OLEBankEV project as a server application.
3. Mark the BankServerOLE service object as an OLE server, as described in "Mark a Service Object as an OLE Server" on page 76.
4. Remove the server partition from all nodes that are not running Windows NT.
5. Make a distribution using autocompile and autoinstall.
6. Start the iPlanet UDS server partition, as described in "Start the iPlanet UDS Partition" on page 85.
7. Using Visual Basic, open the OLEBankEV.vbp file.
8. Make sure that the OLEBankEV project can find the BankEV.frm file by using the Visual Basic File > Add File command.
9. In Visual Basic, use the TOOL > References command to tell the OLE client application the location of the iPlanet UDS service object .tlb file, which is in the `FORTE_ROOT\userapp\olebanke\cl0\` directory.
10. Run the Visual Basic OLE client example. Valid account numbers are 1000, 2000, and 3000.

OLEBankUV

Description OLEBankUV shows how to create a user-visible service object that provides wrapper methods to the methods on other iPlanet UDS service objects. This example then provides a Visual Basic client that shows how an OLE client application could interact with the user-visible service object to use the services provided by the BankServices.BankServer service object.

- olebanuv.pex contains the OLEBankUV project, which defines a class named BankServiceOLEInterface, which defines wrapper methods that in turn invoke methods on the BankServices.BankServer service object. This project also defines a user-visible service object called BankServerOLE.
- All the other files are part of the Visual Basic OLE client.

Pex Files `frame/banksvc.pex`, `extsys/ole/server/olebanuv.pex`.

Mode Distributed only.

Special Requirements This example runs only on a node running Windows 95 and Windows NT. You need to have Microsoft Visual Basic installed on the Windows NT or Windows 95 node and a C++ compiler. You should have autocompile set up.

The Visual Basic clients were written to use Visual Basic Version 4.0. If you are using later versions of Visual Basic, you might need to upgrade the provided Visual Basic components, adjust the following instructions.

➤ **To use OLEBankUV**

1. Import the .pex files, listed above.
2. Configure the OLEBankUV project as a client application, with the BankServerOLE service object in the client partition.
3. Mark the BankServerOLE service object as an OLE server, as described in "Mark a Service Object as an OLE Server" on page 76.
4. Remove the client partition containing the BankServerOLE service object from all nodes that are not running Windows 95 or Windows NT.
5. Make a distribution using autocompile and autoinstall.
6. Start the iPlanet UDS client partition, as described in "Start the iPlanet UDS Partition" on page 85.
7. Using Visual Basic, open the OLEBankUV.vbp file.
8. Make sure that the OLEBankUV project can find the BankUV.frm file by using the Visual Basic File > Add File command.
9. In Visual Basic, use the TOOL > References command to tell the OLE client application the location of the iPlanet UDS service object .tlb file, which is in the `FORTE_ROOT\userapp\olebanku\cl0\` directory
10. Run the Visual Basic OLE client example. Valid account numbers are 1000, 2000, and 3000.

OLESample

Description OLESample uses OLEField, `Olegen`, and iPlanet UDS's implementation of OLE Automation. It uses a Microsoft Chart application (part of Microsoft Graph5.0). The chart is embedded in an OLEField. `Olegen` has been run to create a Graph project. OLE Automation methods are used to access and manipulate objects in the chart.

Pex Files `extsys/ole/msgraph.pex`, `extsys/ole/olesam.pex`.

Mode Standalone or Distributed.

Special Requirements MSWindows3.1 or NT environment, MSGraph5.0.

➤ **To use OLESample**

1. The OLESample .pex files are not imported automatically by the tstapps.fsc script, so you must first import them in the order given above. msgraph.pex was generated by invoking `Olegen`. The following command line was used:

```
-- If you run this, use paths appropriate for your environment.
olegen -it c:\windows\msapps\msgraph5\gren50.olb
-of c:\examples\extsys\ole\msgraph.pex
```

   You can generate your own `msgraph.pex`, to see `olegen` in operation, or you can use the `msgraph.pex` provided in the examples directory.

2. Start the application. Click on the New Graph button. The embedded OLE field will activate a generic Microsoft Graph Chart application.
3. Click on the iPlanet UDS window to deactivate the field.
4. Double-click in the OLE chart field to activate it. Choose Insert and Titles from the Microsoft Graph Chart menu. Choose Chart Title and click the OK button. Change the title if you wish.
5. Click in the iPlanet UDS window to deactivate Microsoft Graph Chart.
6. Click the Rotate Chart button as many times as you like.
7. Click the Change Title button and provide a title of your choice.

8. When you exit the example, the graph with your changes will be saved in the file `olesam.out` in `$FORTE_ROOT/tmp`.

9. Start the application again. This time the Load Saved Graph button will be activated. Click it. The chart it loads will reflect the changes you just made after creating a new chart. You can change the title and rotate the graph again. These changes will be saved when you exit the application.

OutboundExternalConnection

Description OutboundExternalConnection illustrates how to use the ExternalConnection class to initiate a connection. The C program extcon waits for a new connection. OutboundExternalConnection will initiate the connection extcon is waiting for. Once the connection is established, data is read and written. For the read, the iPlanet UDS program makes sure the anticipated number of bytes have been received. For the write, the iPlanet UDS program uses the UseData method on MemoryStream to improve efficiency.

Pex Files `outbound.pex`.

Mode Distributed.

Special Requirements C compiler; C portion of this example will run on NT and Unix platforms; it will not run on Mac, Windows, or VMS.

➤ **To use OutboundExternalConnection**

1. Decide which platform you want to run the C program on, and which platform you want to run the iPlanet UDS program on. Compile the C program `extcon.c` into the executable extcon on the desired platform.

   On most Unix systems, simply use the following command:

   ```
   cc extcon.c -o extcon
   ```

   This will work on the following platforms:

   - AlphaOSF
   - RS6000
   - Solaris
   - Data General

On Sequent, use the following command:

```
cc extcon.c -o extcon -lsocket -linet -lnsl
```

On HP, use the following command:

```
cc +Z extcon.c -o extcon
```

On NT, if you use Visual C to compile extcon.c, make sure to include `wsock32.lib` with your standard Object/Library modules. Also, make sure the application is defined as a console application, not a windows application.

2. Both the iPlanet UDS program and the C program will use a default port number for the listener, unless you supply it as an environment variable. The default port number is 6868. If you need to use another port number, set the environment variable FORTE_EP_REG_PORT_2 to the desired port number in both the environment where you will run the iPlanet UDS program and the environment where you will run the C program.

3. If you want to establish an external connection between the iPlanet UDS program and the C program running on the same machine, you do not need to set an environment variable for the node name. If you want to connect between different machines, you will need to set an environment variable. Set the environment variable FORTE_EP_NODENAME_2 in the environment where you are running the iPlanet UDS program. Set it to the name of the machine where the C program is running.

4. On the machine where you compiled extcon, run it with the w command line option, so that it will wait for a connection:

   ```
   extcon w
   ```

   Extcon will time out after three minutes. If you need more than three minutes to start the iPlanet UDS part of this example, edit extcon.c. Increase the value of DEFAULT_REG_UPTIME and recompile extcon.c

5. Use the file `outbound.scr` to supply the necessary commands to fscript. `Outbound.scr` will import the pex file `outbound.pex`, find the project, run it, and remove the project after the run is complete. `Outbound.pex` must be in the same directory as `outbound.scr`. Use fscript's -i flag to input `outbound.scr` to fscript:

   ```
   fscript -i outbound.scr
   ```

6. Observe the output of both processes. On the iPlanet UDS side, you should see the following results:

```
Attempting to make a connection on port 6868.
Attempting to make a connection on canis.
OutboundConnection: server read got back 14 bytes
Data received.
Outbound connection: close done
```

From the C program, you should see the following lines:

```
Waiting to connect on port 6868.
QString = Tire<EOW1>65psi<EOW2>Inflated<EOW3>
```

XRefTime

Description XRefTime is an example of a TOOL C project, along with a TOOL project that calls the TOOL C project. It is useful for seeing how to use the ExternalRef class to free memory associated with iPlanet UDS objects after those objects have been reclaimed by memory management. The example program MathTime shows how to write C projects. The example program DMathTm is the distributed version of MathTime. The example program AllCType is a reference for how to define and call TOOL C methods with parameters of all C data types at assorted levels of indirection.

Pex Files `extsys/c/xreftime.pex`.

Mode Distributed only.

Special Requirements Access to standard C Runtime Libraries and a C++ Compiler, creation of a working directory, autocompile must be turned on.

➤ **To use XRefTime**

1. Create a working directory where you have read and write permission. Copy the following three files from `$FORTE_ROOT/install/examples/extsys/c` to your working directory: `xreftime.pex`, `xreftime.fsc`, `xreftime.c`. Set the environment variable FORTE_EP_WRKDIR to your working directory.

2. The file `xreftime.pex` contains the C project XRefTimeProject and the TOOL project TestXRefTimeProject. They assume you have access to the standard C runtime libraries. Make sure you know where these are located and what they are called on your system.

3. Edit your copy of `xreftime.pex` so that its ExternalSharedLibs extended property points to the standard C shared library. Search the file for the string '`/usr/shlib/libc`'. Change this string to the correct path and library name for your system.

4. Compile the file `mathtime.c` into an object file called xreftime.o. Under the directory `$FORTE_ROOT/tmp`, create the directory '`examples`', if it isn't there already. Copy the file `xreftime.o` to the `$FORTE_ROOT/tmp/examples` directory.

5. Before completing this step, make sure autocompile is available on your system. If autocompile is not set up, ask your System Administrator to set it up for you. Now run Fscript and enter the following commands:

```
UsePortable
SetPath %{FORTE_EP_WRKDIR}
Include xreftime.fsc
```

6. The xreftime.fsc script will import, distribute, compile, install, and run the XRefTime example. You may want to examine this script to understand the steps involved in linking TOOL code with external C routines.

XML Services Sample (BankServices Project)

Description An application implementing basic banking services that are exported using XML services. The BankServices application provides access to account information for specific accounts, and also allows you to make deposits and withdrawals. This application is based on the BankServices application used in the tutorial provided in *Getting Started With iPlanet UDS*.

In this example, the BankServices application exports the BankMgr class as a service object. It then generates a Java proxy file and a JavaBean file that are used by the provided Java clients to access the exported bank services.

Pex File `xmlsvr/BankServices.pex`.

Script Files The following script files, located in the same directory as the .pex file, automate the building of the BankServices server and run the provided Java client:

```
BankServices.fsc
StartBankServices.esc
StopBankServices.esc
```

Java Client Files The following Java files implement both a command line and a Swing client to the BankServices XML server. The command line client provides hard-coded calls to the server while the Swing client demonstrates an interactive client.

```
BankClient.java
Welcome.java
DisplayAccountDialog.java
NewAccountDialog.java
```

Generated Java Files When you run the example script, the following Java source files are generated:

```
BankSOProxy.java
BankAccount.java
```

These files are placed at:

```
FORTE_ROOT/appdist/envname/bankserv/cln/generic/java/
com/forte/xmlserver/bankservices
```

The example script later invokes a Java compiler to compile the files so they can be used by the Java clients.

Special Requirements Java Development Kit (JDK), XML parser, SOAP library. Refer to the comments in `BankServices.fsc` for information on setting up your environment. Refer to the platform matrix at http://www.forte.com/support/platforms.html for any additional setup requirements.

➤ **To run the BankServices example**

1. Set up your environment according to the comments in `BankServices.fsc`.

   Environment considerations include:

   - PATH set to supported version of JDK
   - CLASSPATH properly configured to access an XML parser and SOAP libraries
   - DISPLAY variable properly set for UNIX systems
   - Java compiler and Java runtime configured in FORTE_JAVACOMP and FORTE_JAVAVM environment variables
   - FORTE_ENVNAME set to your UDS environment name

     For example (depending on your installation), centrale on Windows platforms or CentralEnv on UNIX platforms
   - FORTE_SERVER set to the node in your UDS environment that is hosting the XML server
   - FORTE_PORT set to nt (for Windows platforms only)

2. Start your iPlanet UDS environment and make sure you have a repository server running.

   If the Repository Workshop is open, shutdown the Repository Workshop.

3. Run `BankServices.fsc` as follows:

   ```
   fscript -i BankServices.fsc
   ```

When the server and command client are successfully deployed, the command line client prints the following to standard out:

```
Name:Paul Cezanne
Account Number:1000
Account Balance:300.75

Name:Xavier M. Lawrence
Account Number:4000
Account Balance:777.77
```

When the Swing client is successfully deployed, the following dialog is displayed:

The account numbers 1000, 2000, and 3000 are valid for this example. Additionally, the account number 4000, created by the command line client, is also valid.

Appendix B

Olegen Mapping Conventions

This appendix describes how the `Olegen` utility interprets the interfaces provided by OLE servers and ActiveX controls.

Olegen Mapping Conventions

The `Olegen` utility expects the OLE server or ActiveX control to provide type libraries, either in a file outside the application, or as output when the type libraries are requested from the application.

ActiveX controls are a special type of OLE automation server, so information presented in this section applies to the interfaces for either an OLE server application or an ActiveX control, unless otherwise indicated.

These type libraries are usually generated by compiling a file containing Object Description Language (ODL) statements, which describe the dispatch interfaces available for an OLE server. The `Olegen` utility uses the ITypeLib and ITypeInfo interfaces provided by each type library to access the data type information for each OLE method provided for an OLE server application.

The following sections describe the conventions that the `Olegen` utility uses when mapping OLE automation methods to TOOL methods.

Mapping OLE Automation Interfaces to TOOL Classes

The `Olegen` utility uses the type library or dispatch interface provided by the OLE server to determine how to map the OLE automation interfaces for a Windows application to TOOL classes. If the OLE server provides:

A type library, either as a file or at runtime The `Olegen` utility defines the project name as the name of the type library. The `Olegen` utility maps each dispatch interface defined in the type library to a TOOL class.

Information for only one dispatch interface `Olegen` generates one class for the OLE methods whose interface definitions are provided by the single dispatch interface. The `Olegen` utility might not be able to access information about all OLE methods provided for an OLE server.

No type library The `Olegen` utility generates one class for the OLE methods whose interface definitions are provided by the OLE server. In this case, `Olegen` might not be able to access information about all OLE methods provided for an OLE server.

No type information `Olegen` cannot generate any TOOL classes.

Mapping ActiveX Interfaces to TOOL Classes

The following sections describe the conventions that the `Olegen` utility uses when mapping ActiveX control methods to TOOL methods.

The `Olegen` utility uses the type library or dispatch interface provided by the ActiveX control to determine how to map the OLE automation interfaces for a Windows application to TOOL classes. The ActiveX control provides:

A type library, either as an .ocx file or at runtime The `Olegen` utility defines the project name as the name of the type library. The `Olegen` utility maps each dispatch interface defined in the type library to a TOOL class.

Type library with information for only one dispatch interface `Olegen` generates one class for the OLE methods whose interface definitions are provided by the single dispatch interface.

No type library The `Olegen` utility gets information directly from the ActiveX control and generates one class for all the methods provided by the ActiveX control.

No type information `Olegen` cannot generate any TOOL classes.

The `Olegen` utility expects the ActiveX control to provide type libraries in one of the following ways:

- in a file outside the control, usually in a file with an .ocx extension
- as output when the type libraries are requested from the control

Mapping Data Types in TOOL

The OLE interfaces provided by some Windows programs sometimes do not provide enough data type information to strongly type data the way TOOL requires. Therefore, the `Olegen` utility uses an Object subclass called `Variant`, and its subclasses, to permit the data type of a parameter to remain unspecified until run time. The `Variant` class and its subclasses are defined in the OLE library. For more information about the `Variant` class and its subclasses, see the iPlanet UDS online Help.

When the `Olegen` utility can determine the mechanism for the parameter, but not the specific data type, the `Olegen` utility uses the `Variant` class. The `Olegen` utility also sets an attribute of the `Variant` class called Mechanism to define the usage intended by the parameter. Parameters that are variant objects in the OLE server's methods are mapped as input parameters of the `Variant` class or its subclasses. `Olegen` then sets the Mechanism attribute of the Variant class to `VARIANT_IN`, `VARIANT_OUT`, `VARIANT_INOUT`, or `VARIANT_RESULT` to define the usage intended by the parameter, as described in the iPlanet UDS online Help.

If the `Olegen` utility can determine the data type and usage for a required parameter, it maps the data type to a TOOL data type, as shown in the following example:

```
input "param1" : Framework.integer;
```

If the `Olegen` utility cannot determine the data type or usage for a parameter, or if the parameter is optional, then `Olegen` maps the parameter using the `Variant` class or one of its subclasses, as shown in the following example for an input parameter:

```
input "Index" : Variant = NIL,
```

If the `Olegen` utility cannot determine the data type or the usage of the parameter, as in the preceding two cases, and declares the parameters using the `Variant` class, you must be able to determine the required data type from the documentation for the interface for the Windows application.

To invoke a method with parameters of an unknown type or usage, you must instantiate an object of the `Variant` subclass for the correct data type, for example, VariantI2, and set the Mechanism attribute of the object to the correct usage, for example, `VARIANT_INOUT`. The following example shows these steps:

```
-- Create a VariantI2 object that is an input output parameter
ParmValue : VariantI2 = new(Mechanism = VARIANT_INOUT, Value = 17);
```

At runtime, when your TOOL application invokes this method in the OLE server, the OLE server checks the parameter type to ensure that it matches the required data type. If the parameter does not match the required data type, the OLE server returns error information, and iPlanet UDS raises an OLEInvokeException. For more information about OLEInvokeException, see the iPlanet UDS online Help.

Mapping Return Values of Methods

OLE methods can be overloaded so that the only differences among methods is the data type of the return value. However, TOOL differentiates methods only based on the parameters declared within the parameter list, not on the return value.

Therefore, the `Olegen` utility generates an extra parameter, called `_result`, in all the TOOL methods. `_result` represents a return value from the mapped OLE server method. The following example shows how the `Olegen` utility includes the `_result` parameter in the parameter list when it generates the `.pex` file:

```
method "ApplyDataLabels" // function 1
  (
  input "Type" : OLE.Variant = NIL,
  input "LegendKey" : OLE.Variant = NIL,
  input _result : OLE.Variant = NIL
  ) : OLE.Variant;
```

In the above example, the `_result` parameter is optional and of the `Variant` class. The following example shows another method also called "`ApplyDataLabels`" that always returns an integer value. iPlanet UDS can recognize the difference between the two methods because the return value is included in the parameter list as the `_result` parameter.

```
method "ApplyDataLabels" // function 1
  (
  input "Type" : OLE.Variant = NIL,
  input "LegendKey" : OLE.Variant = NIL,
  input _result : integer
  ) : OLE.Variant;
```

Mapping Optional Parameters in Methods

When the `Olegen` utility generates a TOOL method from an OLE method that has optional parameters, the optional parameters are assigned NIL values, as shown in the following example:

```
InputParameter : VariantInteger = NIL;
```

Mapping Names That Are iPlanet UDS Reserved Words

When the `Olegen utility` maps OLE interface methods to TOOL methods, the `Olegen` utility surrounds the names used in the OLE interface method with double quotation marks, as shown in the following example:

OLE interface method	TOOL method

object.**dialogs [(***index***)]**	`method "Dialogs"`
	`(`
	`input "Index" : Variant = NIL,`
	`input _result : OLE2Interfaces.Variant = NIL`
	`) : OLE2Interfaces.Variant;`

When you import the `.pex` file, iPlanet UDS removes the quotation marks from the methods. However, when you use a method whose name or whose parameters' names are TOOL reserved words, then you need to specify double quotation marks around the names that are reserved words. To see the list of TOOL reserved words, see *TOOL Reference Guide*.

The following example shows how you use a method that uses TOOL reserved words as names:

```
method "Input"
   (
   input "param1" : Framework.i2,
   input "param2" : Framework.i2,
   input _result : OLE2Interfaces.VariantString = NIL
   ) : Framework.string;
```

After you import the `.pex` file containing this method definition, you can use the method in your code, as shown in the following example:

```
OLEobject : OLEclass = new;
OLEobject.CreateUsingProgID('Word.Basic');
RetrievedString : string;
RetrievedString = OLEobject."Input"(param1=1, param2=3);
```

Mapping ActiveX Control Events to iPlanet UDS Events

The iPlanet UDS event names have the same names as the control's events, except that they start with an underscore character (_). For example, if the control can send the Click event, the `Olegen` utility generates an iPlanet UDS event called _Click. Similarly, if a control's event has the name _Click, then the corresponding iPlanet UDS event's name is __Click (two underscores).

The generated iPlanet UDS event has the same signature as the control's event, except that all parameters are input parameters, even when some of the parameters of the control's event are output or input output.

The `Olegen` utility also generates a method for each method that maps to ActiveX control events. The name of the method is exactly the same as the name of the event for the custom control.

For example, the FourDir custom control defines an event called Click. This event maps to the _Click event and the `Click` method in the ActiveX interface class.

Index

SYMBOLS

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X