Connect:Direct®
for Windows SDK

Programmer's Guide

Version 4.4

Sterling Commerce
An IBM Company

Connect:Direct for Windows SDK Programmer’s Guide
Version 4.4

First Edition

CDWSDKPG707

Contents

Chapter 1

Connect:Direct for Windows SDK

The *Connect:Direct for Windows SDK Programmer's Guide* provides system programmers and system developers with the information needed to extend the capabilities of the Connect:Direct environment.

Overview

Use the Software Development Kit (SDK) to extend an application to include the automated file transfer capabilities of Connect:Direct for Windows. SDK uses a 32-bit interface for C and C++ as well as an OLE automation server for Visual Basic applications. SDK also provides ActiveX controls for Submit Process and Select Statistics commands. The tools available in SDK include:

- C API functions—Standard and registry API functions. The standard functions allow you to connect to a Connect:Direct node, execute Connect:Direct commands, manage command response data, and retrieve error information. The Registry API functions store and retrieve client connection information to and from the Registry. The C API is implemented using the C++ Classes.
- C++ Class interface—Provides the foundation for the other Connect:Direct interfaces and provides Visual C++ programmers an object-oriented interface to Connect:Direct.
- ActiveX control interface—Uses the CDSubmit and CDStatistics functions to submit Processes to the server and display statistics from the statistics database.
- Direct Automation Servers—Provides an automation wrapper around the Connect:Direct SDK C++ classes. They provide direct automation support for languages like Visual Basic. The Connect:Direct Automation Servers provide the following primary classes that map directly to the CDNode, CDProcess, and CDStatistics classes in the Connect:Direct SDK C++ classes:
- User exits—Provides a way to customize Connect:Direct operations. User exits are user-defined dynamic link libraries (DLLs) that are loaded and called when the user exit is enabled through an initialization parameter. Two user exits are provided: one for enhanced security and one for automated file opening.

Before you can use the SDK tools, you must run the Client Connection Utility initially to configure server access information, such as TCP/IP information.

Connect:Direct for Windows Documentation

See *Connect:Direct for Windows Version 4.4.00 Release Notes* for a complete list of the product documentation.

About This Guide

The *Connect:Direct for Windows SDK Programmer's Guide* is for programmers and network operations staff who install, configure, and maintain the Connect:Direct for Windows product.

This guide assumes knowledge of the Windows operating system, including its applications, network, and environment. If you are not familiar with the Windows operating system, refer to the Windows library of manuals. Likewise, users should be familiar with C, C++, or Visual Basic programming language. If you are not familiar with the C or C+ programming language, refer to the relevant documentation.

Task Overview

The following table directs you to the information required to perform the tasks documented in this guide:

Task	**Reference**
Using the Connect:Direct Client Connection Utility to view, edit, update, import, export, and print the Connect:Direct for Windows client-to-node connection settings in the Windows Registry.	Chapter 2, *Editing Connection Settings*
Understanding the purpose and format of the Connect:Direct C API functions.	Chapter 3, *Applying the C Applications Programming Interface*
Understanding the purpose and format of the Connect:Direct C++ Class interface	Chapter 4, *Applying the C++ Class Interface*
Applying the Connect:Direct ActiveX Controls	Chapter 5, *Applying the ActiveX Control Interface*
Applying the Connect:Direct Automation Servers	Chapter 6, *Applying the Automation Servers*
Applying user exits to enhance security and automate file opening	Chapter 7, *Enhance Security and Automate File Opening with User Exits*
Refernce information on C and C++ structures and return codes	Appendix A, *Structure Types* Appendix B, *Return Codes*

Chapter 2

Editing Connection Settings

Before you begin using the SDK to create your own programs, you must create connection settings for each user. Two methods are available to create local node definitions. You can use either Connect:Direct Requester or the Client Connection Utility. If you want to use Connect:Direct Requester, refer to the *Connect:Direct for Windows System Guide* for instructions. This chapter provides instructions on using the Client Connection Utility.

About the Client Connection Utility

The Connect:Direct for Windows client software uses the Microsoft Windows Registry to store its configuration information. The Connect:Direct Client Connection Utility allows you to update the connection settings within the Registry.

Caution: Use the Connect:Direct Client Connection Utility to update any Registry settings rather than editing them directly.

You can view, edit, and update Connect:Direct for Windows connection settings in the Windows Registry with the Client Connection Utility. These settings enable communication between the user interfaces and the Connect:Direct server. You can set up and update connection settings in the following ways:

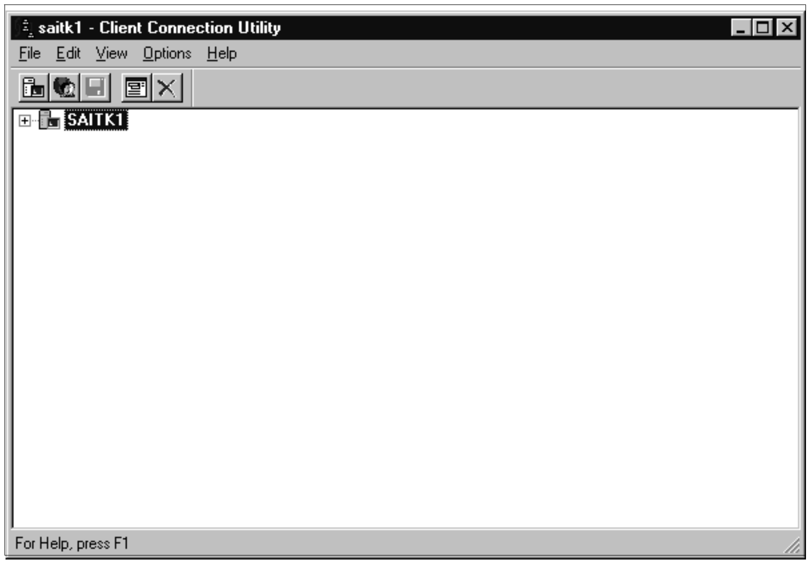
- Add and delete a node
- Add and delete a user
- Configure node and user properties
- Define a default node or user

To facilitate updating connection settings on multiple servers, you can import and export connection settings using the Client Connection Utility. After you configure the connection for a server, you can export the server's settings for use on other servers. You can then import the settings into the target server's Registry. You can also print connection settings.

Starting the Client Connection Utility

To start the Client Connection Utility:

1. Click **Start**, and then point to **Programs**.
2. Click the **Sterling Commerce Connect Direct v**4.4.00 program.
3. Select **CD Client Connection Utility**. The Client Connection Utility main window is displayed.


saitk1 - Client Connection Utility
File Edit View Options Help
SAITK1
For Help, press F1

Understanding the Tool Bar

The Client Connection Utility toolbar provides the following icons to perform frequently used actions:

Select	To
	Create a new node.
	Create a new user.
	Save Registry settings.
	Configure highlighted node or user settings.
	Delete a node or user.

Adding and Deleting a Node Connection Definition

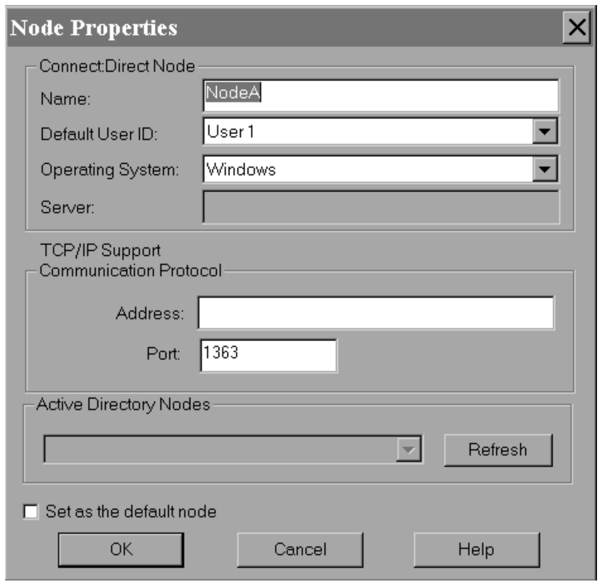
The Connect:Direct Client Connection Utility enables you add new Connect:Direct nodes and identify the properties of the nodes, such as node name, TCP/IP address, and port number. These properties establish a node so you can access it from Connect:Direct Requester or the Command Line Interface (CLI).

You can also use the Client Connection Utility to delete existing nodes.

Adding a Node

To add a new Connect:Direct for Windows node:

1. From the **File** menu, select **New Node** to display the Node Properties dialog box:



Node Properties

Connect:Direct Node
Name: NodeA
Default User ID: User 1
Operating System: Windows
Server:

TCP/IP Support
Communication Protocol
Address:
Port: 1363

Active Directory Nodes

2. To add a node registered in the Active Directory:
   a. Select Windows in the **Operating System** field.
   b. Select the node to add from the Active Directory Nodes drop down box.

      The name, address, and port fields are automatically updated with information from the Active Directory list.

      Note: Click **Refresh** to update the address and port stored on the local computer with the values from the Active Directory listing.

3. To add a node that is not registered in the Active Directory, do the following:
   a. In the **Name** field, type the name of the Connect:Direct node you want to add.
   b. If necessary, change the operating system value in the **Operating System** field.

c. In the **Address** field, type the TCP/IP address of the new node.

d. The **Port** field automatically defaults to 1363; if necessary, type in a different port number.

4. To specify the new node as the default node, click the **Set as the default node** check box.
5. Click **OK** to save your settings and close the **Node Properties** dialog box.
6. From the **File** menu, select **Save** to save the new settings.

Note: Changes made to the node settings are *not* written to the Registry until you select **Save**.

Deleting a Node

To delete a node:

1. In the Client Connection Utility main window, select the node you want to delete.
2. Select **Delete** from the **Edit** menu.
3. Click **Yes** to confirm the deletion.
4. Select **Save** from the **File** menu to delete the node.

Note: Changes made to the node settings are *not* written to the Registry until you select **Save**.

The node is no longer displayed in the Client Connection Utility window.

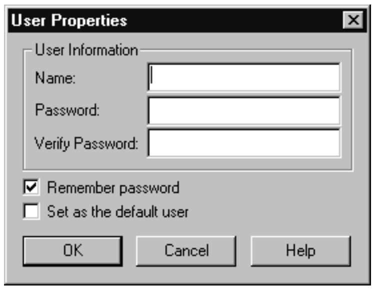
Adding and Deleting a User Connection Definition

Use the Connect:Direct Client Connection Utility to set up new users for a node by specifying user properties, such as the user name and password. These properties establish users of the node who can access the server through Connect:Direct Requester or the CLI.

Adding a User

To add a new Connect:Direct user:

1. In the **Client Connection Utility** main window, select the node where you want to add a new user.
2. From the **File** menu, select **New User** to display the User Properties dialog box.



3. Type information into the following fields:
   - **Name—**type the name of the new user. Either type the user name as defined in the Windows setup, such as lmore or type a fully qualified user name in the UPN format, such as lmore@adtree.stercomm.com
   - **Password—** type the password defined for the user.
   - **Verify Password—**retype the password defined for the user.
4. Click the **Remember password** check box to automatically reload the password when you attach as this user.
5. Click the **Set as the default user** check box if you want the new user to be the default user for the node.
6. Click **OK** to save the settings and close the **User Properties** dialog box.
7. If the verification password you typed does not match the initial password, you receive a message indicating the passwords do not match when you click **OK**. Retype the verification password and click **OK**.
8. From the **File** menu, select **Save** to save the settings.

Note: Changes made to node settings are *not* written to the Registry until you select **Save**.

Deleting a User

To delete a user from the node:

1. If the user names are not displayed, click the plus (+) sign next to the node containing the user you want to delete.
2. Select the user you want to delete.
3. From the **Edit** menu, select **Delete**.
4. Click **Yes** to confirm the deletion.
5. From the **File** menu, select **Save** to save the new configuration.

Note: Changes made to node settings are *not* written to the Registry until you select **Save**.

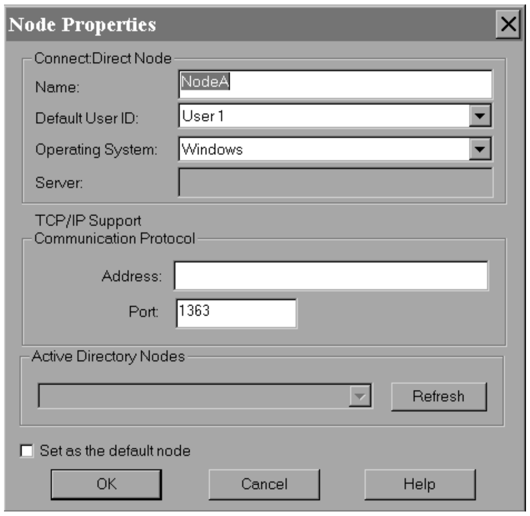
Updating Node and User Properties

The Connect:Direct Client Connection Utility lets you update node and user properties to maintain accurate information.

To update node properties:

1. Do one of the following:
   - To update a node, highlight the node you want to configure.
   - To update a user properties, highlight the user you want to configure.

2. From the **File** menu, select **Properties** to display the Node Properties dialog box:



Connect:Direct Node
Name: NodeA
Default User ID: User 1
Operating System: Windows
Server:
TCP/IP Support
Communication Protocol
Address:
Port: 1363
Active Directory Nodes
☐ Set as the default node

3. Select the fields you want to edit and make the appropriate changes.
4. Click **OK** to save your settings and return to the **Node Properties** dialog box.
5. From the **File** menu, select **Save** to save the settings.

Note: Changes made to node settings are *not* written to the Registry until you select **Save**.

Defining a Default Node or Default User

The Connect:Direct Client Connection Utility allows you to define a default node or default user. The default node and user will be used by the Connect:Direct Requester and the CLI.

To define a default node or user:

1. Take one of the following actions:
   - To define a default node, highlight the node.
   - To define a default user, highlight the user.
2. From the **Options** menu, select **Set as Default** to set the default node or user.
3. From the **File** menu, select **Save** to save the settings. The default node or user is displayed in the main Client Connection Utility window as bold text.

Note: Changes made to node settings are *not* written to the Registry until you select **Save**.

Importing and Exporting Registry Settings

The Connect:Direct Client Connection Utility allows you to import and export connection settings to a file. These settings can be saved and used on another computer or node.

Importing Registry Settings

To import Registry settings:

1. Select the node in which to import the Registry settings.
2. From the **File** menu, select **Import**. A message informing you that all settings will be lost is displayed.
3. Click **Yes**. The **Open** dialog box is displayed.

Note: Importing a Registry settings file causes all current changes to the selected node to be lost if they have not been saved.

4. Select the Registry settings file you want to import (.REX extension) and click **OK.** The imported Registry settings are applied to the node you selected.
5. From the **File** menu, select **Save** to save the settings.

Note: Changes made to node settings are *not* written to the Registry until you select **Save**.

Exporting Registry Settings

To export Registry settings:

1. From the **Client Connection Utility** main window, select the node containing the Registry settings you want to export.
2. From the **File** menu, select **Export**. The **Save As** dialog box is displayed.
3. Name the exported Registry file with a REX extension and click **OK**. The Registry settings in the file can now be imported to another computer or node.

Printing Registry Settings

The Connect:Direct Client Connection Utility allows you to generate a report of Registry settings. To print the Registry settings report:

1. To preview the Registry settings report before printing it:
   a. Select **File > Print Preview**.
   b. Click **Zoom In** to enlarge the text and read the report.

2. To print the report:
   a. Select **File > Print** to display the **Print** dialog box.
   b. If necessary, select the printer.
   c. Click **OK**. A report of all Registry settings is generated.

Note: Additional node detail is provided if the node has been used at least once by the client software.

Chapter 3

Applying the C Applications Programming Interface

This chapter describes the purpose and format of the Connect:Direct C API functions. The Connect:Direct C applications programming interface consists of Standard and Registry API functions. The Standard API functions connect to a Connect:Direct node, execute Connect:Direct commands, manage command response data, and retrieve error information. The Registry API functions store and retrieve client connection information to and from the Registry. The C API is implemented using the C++ Classes. This interface is used by C programmers.

Understanding Standard C API Functions

This section provides an overview of the Standard C API functions and their responses.

Using Handles

Handles simplify object and memory management by referencing a particular object. Pass a handle to an API to uniquely identify an object. The Connect:Direct C API uses the following types of object handles to return node, Process, statistics, message, and trace information:

✦ Node Handles—Represent the Connect:Direct node that is the target of the operation. It is a virtual connection to a Connect:Direct node. The node handle is a special type of object handle; it holds information about the node, but does not return data from the node.

  A node handle is created by calling the **CdConnect()** function and passing in the node name, user ID, password, and protocol within a NODE_STRUCT structure. After you finish with a node handle, you call the **CdCloseHandle()** to close it. Closing the handle releases the virtual connection and any internal resources associated with it. The node handle is no longer valid on subsequent operations.

  Note: You are responsible for closing the node handle and for releasing any resources that you allocate.

- Process Handles—Handles returned from a **submit** command or from a Process object, which is created when a select process, change process, or delete process command is executed. The following example demonstrates the **select process** command returning a Process:

```
if (CdExecuteCommand (hNode, “SELECT PROCESS”, &hProc))
{
     if (CdGetProcRec(hProc, &Proc))
     {
     printf("%d %s/n", Proc.ProcessNumber, Proc.ProcessName);
     }
}
```

- Statistic Handles—Statistics objects that are returned after a **select statistics** command is executed.
- Message Handles—Message objects that are returned when a **select message** command is executed.
- Trace Handles—Trace objects that are returned when a **traceon** or **traceoff** command is executed.

Waiting for a Process

Use the following function to wait for a Process:

- **CdWaitOnProcess()**—Use this function to serialize Connect:Direct Process execution. This function blocks the calling thread until the specified Process is no longer in the TCQ. It takes a Process handle that contains references to the target Process object. Any Process object handle can enable you to specify Processes to wait on. Use this method to wait on a Process returned from a **submit** command and any Process returned by the **select process** command.

Retrieving Error Text

Use the following functions to retrieve error text:

- **CdGetErrorText()—**Call this function to translate return code values into messages that explain the error. This helps the user understand the error message and provides a method for logging meaningful trace messages within an application.
- **CdGetDetailedError()—**Use this function to retrieve messages one at a time until CD_ENDOFDATA is returned. This call fills in the MESSAGE_STRUCT structure with a detailed error message for node, parser, and connection errors. The messages are erased upon entry to any other API to prepare for other potential errors.

Blocking

The C Application Programming Interface is synchronous; when an API that performs a complex function (such as the **CdConnect()** or **CdExecuteCmd()** functions) is called, the caller’s thread is blocked until the request is completed or until a failure occurs. The caller’s thread blocks while waiting for other threads to finish the request.

If the **CdConnect()** function is called from a Windows application, it should not called from the primary user interface (UI) thread. Calling the function from the UI thread causes the user interface of the program to run slowly.

Compiling and Debugging

When you are ready to compile the program created with the API, include the CDCAPI.H header file. Including the CDCAPI.H file in your project automatically links a program with the appropriate import library. Debug configurations link with the CDCAPID.LIB and release configurations link with the CDCAPI.LIB.

The CDCAPI.LIB and CDCAPID.LIB files contain the following information:

- ✦ Name of the DLL to dynamically load at run time.
- ✦ Definitions of all exported functions. This is used by the linker to resolve all calls to the CDCAPI.DLL.

When the program runs or the DLL is loaded, the appropriate CDCAPI.DLL is loaded. The CDCAPI.DLL is dynamically loaded when a release configuration is executed, and the CDCAPID.DLL is dynamically loaded to support debug configurations.

The C APIs are based on the core C++ APIs. This required API layer is contained in CDCORE.DLL (or CDCORED.DLL if compiling for debug mode). The appropriate core DLL must be in your path for the C APIs to work properly.

Turning Tracing On

The Output window of the Microsoft Visual Studio displays trace messages. Use the following parameters to activate tracing:

Parameter	Description
CdGetTraceFlags(unsigned int* ***pgrfTrace***);	Retrieves the current trace settings for the Connect:Direct API.
CdSetTraceFlags(unsigned int ***grfTrace***);	Sets new trace settings for the Connect:Direct API.
CdSetTraceFile(LPCTSTR ***pszFilename***);	Provides a file name to the tracing facility. If a file is given, trace messages are written to the Output window and to the specified file.

Understanding Registry C API Functions

The Connect:Direct Registry C API functions store and retrieve client connection information to and from the Registry. This connection information is used by the **CDConnect()** function to connect to a Connect:Direct node. This information is stored in the NODE_STRUCT structure.

Using the Standard API Functions

When using the C APIs, it is not necessary to follow a sequence. Refer to the *C API Reference Guide* for a list of standard API functions. All handles are self-contained and are not dependent on other handles.

You must call the **CdConnect()** function to create a valid node handle. This handle is required to execute commands on the node using the **CdExecuteCommand()** function. When all commands are executed, the handle obtained from the **CdConnect()** function is no longer required. The handles returned by the **CdExecuteCommand()** function are completely self-sufficient. When the handle is no longer needed, call the **CdCloseHandle()** function to release the unused resources. Following is an example of an execution sequence:

```
CdConnect
     CdExecuteCommand
          (Any API used to access either handle)
     CdCloseHandle
CdCloseHandle
```

Understanding API Errors

Most of the C APIs return an integer value to indicate the type of error encountered while executing the API. The **CdGetError()** API returns a text string that gives a general description of the error as indicated by the return code value passed into the API. Refer to Appendix B, *Return Codes* for a description of the available return codes. For information about an error, repeatedly call **GetNextMsg()**.

Viewing Sample Programs

Refer to the documentation CD-ROM directory, CDSDK\Samples for the C, C++, and Visual Basic sample code. The sample code contains the following:

✦ The CSample1.C sample program demonstrates how to connect to a node, execute a command, and view the data returned by the node.

✦ The CSample2.C sample program demonstrates a more complex transaction of connecting to a node, submitting a Process, waiting for completion, and requesting statistics for the Process.

✦ CPPSamp1

✦ CPPSamp2

✦ VBAuto

✦ VBStat

✦ VBSubmit

✦ VBSubmit2

Chapter 4

Applying the C++ Class Interface

This chapter describes the purpose and format of the Connect:Direct C++ Class interface. The Connect:Direct C++ Class interface provides the foundation for the other Connect:Direct interfaces and provides Visual C++ programmers an object-oriented interface to Connect:Direct.

Compiling and Debugging

To use the Connect:Direct C++ interface, include the CDSDK.H header file. Including the CDSDK.H file in a project automatically links the program with the appropriate import library. Debug configurations link with the CDCORED.LIB, and the release configurations link with the CDCORE.LIB

Note: You do not need to add the LIB to the LINK section of the project or makefile.

The CDCORED.lib and CDCORE.lib files contain the name of the DLL to dynamically load at run time and class definitions for the linker to resolve the Connect:Direct SDK symbols included in the CDSDK.H file. When a program executes or a DLL is loaded, the appropriate CDCORE.DLL is loaded. Applying.DLL is dynamically loaded when a debug configuration is executed and to support a release configuration.

Turning Traces On

If you want to activate tracing, use the following class methods:

Note: You do not have to create a CDNode to use these methods. Tracing does not relate to nodes directly, but to the Connect:Direct SDK. The methods are placed in the CDNode because it is the central class to everything in the Connect:Direct SDK.

Class Method	Description
CDNode::GetCDSDKTraceFlags();	Returns the current trace settings for the Connect:Direct SDK.
CDNode::SetCDSDKTraceFlags(DWORD bmFlags);	Sets new trace settings for the Connect:Direct SDK.
CDNode::SetCDSDKTraceFilename(LPCTSTR pszFilename);	Provides a file name to the tracing facility. If a file is provided, trace messages are written to the Output window and also to the file.

Manipulating Nodes

The Connect:Direct Component Group classes represent Connect:Direct entities and provide methods to manipulate an object to generate changes on the Connect:Direct node. Use the following classes to manipulate nodes:

Class	Description
CDNode	Contains the high-level Connect:Direct functionality. It returns network map, initialization parameters, and translation table information as well as User and Proxy objects that maintain node information and execute command objects.
CDUser	Contains the user functional authority information. Use to add, delete, and update functional authorities on the Connect:Direct node, including Network map Access Flags, Command Access Flags, Control Flags, Process Statement Flags, and default directories.
CDProxy	Contains the Connect:Direct proxy information. Use to add, delete, and update proxy information on the Connect:Direct node. The remote user proxy contains information for operations initiated from a remote Connect:Direct node and defines relationships between a remote node and local user IDs.
CDTranslationTable	Contains and maintains the translation table information that translates data being sent to other nodes and provides methods for setting and retrieving translation information.
CDTrace	Holds the trace criteria. It contains all the fields returned from the node with the TRACEON command, with no parameters and provides access methods for all of the Trace fields.
CDNetmapNode	Contains the network map node information.
CDNetmapDesc	Contains the description for a network map node.
CDNetmapPath	Contains the network map path information.
CDNetmapMode	Contains the network map mode information.

When using the C++ Class interface, no sequence must be followed when using the C++ classes. All objects are self-contained and are not dependent on any other classes when fully constructed. Each object's constructor is different and some of the objects require another object to be built successfully.

The first and most important class is the CDNode class. This class is the first one to use when interacting with any Connect:Direct node.

While the only prerequisite for constructing a class is the creation of the objects needed by the constructor, the following example shows a possible sample execution sequence:

```
CDNode creation
     CDSelectProcCommand creation
               CDProcIterator creation
                    (Use the data)
               CDProcIterator destruction
     CDSelectProcCommand destruction
CDNode destruction
```

The Connect:Direct CDNode class serves as the virtual Connect:Direct node. It enables you to manipulate and send commands to the actual Connect:Direct node. You manipulate this object through the use of the CDNode methods and issue commands to the node using Command objects. Calling these methods and using the objects sends KQV streams to the physical Connect:Direct node. See the *C++ API Reference Guide* for more information.

Creating an Object to Connect to the Connect:Direct Node

The name of the Connect:Direct node and the connection information is set at object creation time using the CDNode constructor. If a parameter is not supplied (NULL pointer), the default value for that parameter is read from the Registry. During construction, the CDNode object attempts to connect to the physical Connect:Direct node, using the protocol information contained in the Registry. If the connection fails, the CDConnectionException is returned. If the connection is successful, but the logon is denied by the server, a CDLogonException is returned.

The CDNode object creates and removes the connection to the Connect:Direct node as needed. Connections are shared and reused as different requests are made. The following section of the class definition displays the methods to construct a CDNode object and methods to retrieve node information:

```
// Constructor for CDNode
CDNode(LPCTSTR szName=NULL, LPCTSTR szUserid=NULL,
        LPCTSTR szPassword=NULL, int nProtocol=CD_PROTOCOL_TCPIP);
CDNode(LPCTSTR szFilename);
CDNode(const CDNode &Node);
~CDNode();
//Node Information Methods
const CString GetName() const;
LPCTSTR GetCDName() const;
LPCTSTR GetUserid() const;
LPCTSTR GetServer() const;
int GetProtocol();
```

The following two examples illustrate two different methods for creating a CDNode object. The first method creates the CDNode object locally on the stack. The second example creates a dynamic allocation of a CDNode object from the stack. Both methods then execute a SELECT PROCESS command using the CDNode object.

```
{
   CDNode MyNode(“MYNODE”, “MYUSERID”, “MYPASSWORD”);
   CDSelectProcCmd cmd;

   //Execute the “SELECT PROCESS” command
   CDProcIterator it = cmd.Execute(MyNode);

}

{
   CDNode *pNode = new CDNode(“MYNODE”, “MYUSERID”, “MYPASSWORD”);
   CDSelectProcCmd cmd;

   //Execute the "SELECT PROCESS" command
   CDProcIterator it = cmd.Execute(pNode);

   delete pNode;
 }
```

Managing Connections

The CDNode class creates and deletes connections to the Connect:Direct node as needed and deletes the connections if they are idle for a specified period of time. The connections are stored in an array and are created and assigned by the CDNode object when a command requests a connection to the physical node. Connections are reused when they are idle and are deleted if they remain idle for an extended period of time. Because each connection consumes resources on both the client and the server, use them as efficiently as possible. The DisconnectAll member function is used to disconnect all connections to all nodes.

Viewing Information

The Connect:Direct Record Group Classes contain related information about Processes, statistics, messages, and users. Use the following classes to obtain information:

Class	Description
CDProcess	Contains all of the Process criteria information returned from a SUBMIT or SELECT PROCESS command after a Process is submitted. You can submit a Process for execution using one of the following methods: Create a **CDSubmitCmd** object and initialize the parameters. Next, call the CDSubmitCmd::**Execute()** method and specify the CDNode object to run on. Call the **CDNode::Submit()** method and specify the text of the Process. This method internally creates the **CDSubmitCmd** object and calls the **Execute()** method.
CDStatistic	Provides two methods for holding statistics information. **GetAuditField()** Method**—**Because audit data is optional, and different records have different KQV keys, use a single method to access the data. To retrieve a value, call **GetAuditField()**, passing the KQV key for the desired field. The **GetAuditMap()** function retrieves all audit fields defined in the current record. An MFC CMapStringToString object maps from KQV keywords to the corresponding values. This method enables you to view each association in the map to determine what audit fields are available and to ask the map for the value of the given field.
CDMessage	Holds information about a specific message that is retrieved from the Connect:Direct node.
CDUser	Holds the user functional authority information to add, delete, and update functional authority information on the Connect:Direct node.

Controlling the Return of Information

Commands and methods store multiple items in a iterator. The iterator provides methods to enumerate through each returned object.

Understanding Iterators

Commands that retrieve a single record from the server block the calling thread in the **Execute()** method until the data arrives. The data is then put into a record object and returned. Other commands, like **select statistics**, can potentially return hundreds of records. If the **Execute()** method blocks until all records are returned, it can take longer to receive any feedback. If the records are all returned in one large block instead of being consumed at one time, the computer slows down.

To solve these problems, commands that potentially retrieve multiple records return an iterator object as soon as the first record arrives. As data is returned, a background thread automatically appends to the iterator. The iterator has a connection to the server and the command object is not involved. This method allows you to process records as they arrive. The following example demonstrates the **select process** command returning a process iterator:

```
CDSelectProcCmd cmd;
CDProcIterator it = cmd.Execute(node):
```

Accessing Iterator Records

The iterator keeps an internal list of all records returned from the server. Use the following commands to control iterator records:

✦ **HasMore()—**Call this method to determine if any records are available in the list.

> **Note:** You must always call **HasMore()** before calling **GetNext()**. It is not legal to call **GetNext()** if there are no records.

✦ **GetNext()**—If **HasMore()** returns TRUE, obtain the next record in the list using this command. It removes the next record from the list and returns it.

When all records are received from the server, the server notifies the iterator that the command is complete. After all records are removed using **GetNext(), HasMore()** returns FALSE.

If the iterator's list is empty, but the server has not notified the iterator that the command is complete, the iterator cannot determine whether there are more records. In this case, **HasMore()** blocks until more records are received from the server or a completion notification is received. Only then can the iterator return TRUE or FALSE.

The following is an example of accessing statistics records using an iterator:

```
CDSelectStatCmd cmd;
CDStatIterator it = node.Execute (cmd);
while (it.HasMore())
{
     CDStatistic stat = it.GetNext();
     // use the statistics object
}
```

Executing Connect:Direct Commands

The Connect:Direct Command Group classes execute Connect:Direct commands against Connect:Direct nodes. Use the following Command Group classes to execute commands:

Class	Description

CDCommand	The base class for all Connect:Direct command objects. It wraps the parser within a class and enables methods for data manipulation. Each derived class provides an **Execute()** method to execute the command and return the resulting data or object.
	If the result is several items, the command object returns a iterator object that holds the data. The following CDCommand class definition shows the type of methods available in this class:
	`class`
	`CDCommand`
	`{`
	`public:`
	`   // Constructor for CDCommand`
	`   CDCommand(LPCTSTR pCommand=NULL);`
	`   virtual ~CDCommand();`
	
	`   virtual void ClearParms();`
	`   void SetCommand(const CString& strCmd);`
	`   virtual CString GetCommand() const;`
	`   virtual CString GetKQV() const;`
	`   // Execute() methods are provided by each`
	`   // derived command class.`

CDSelectStatCmd	Derived from the CDCommand base class, it enables you to set the SELECT STATISTICS parameters. When you call the **Execute()** method, an iterator data object is dynamically created and attached to the connection assigned by the CDNode object to execute the command.

CDSelectProcCmd	Derived from the CDCommand base class, it enables you to set the SELECT PROCESS parameters. When you call the **Execute()** method, the CDProcIterator object is created dynamically and attached to the connection assigned to execute the command.
	The following example demonstrates the CDSelectProcCmd class:
	
	`CDSelectProcCmd cmd;`
	`CDProcIterator it = node.Execute(cmd);`
	`while (it.HasMore())`
	`{`
	`   CDProcess proc = it.GetNext();`
	`   // use the process`
	`}`

CDChangeProcCmd	Derived from the CDCommand base class, it enables you to set the CHANGE PROCESS parameters. When the **Execute()** method is called, an iterator data object is dynamically created and attached to the connection assigned to execute the command. A CDProcIterator is attached to the iterator data and returned from the **Execute()** method.
CDDeleteProcCmd	Derived from the CDCommand base class, it enables you to set the DELETE PROCESS parameters. When the **Execute()** method is called, a CDProcData object is dynamically created and attached to the connection assigned to execute the command. A CDProcIterator is attached to the iterator data and returned from the **Execute()** method.

Class	Description
CDSelectMsgCmd	Derived from the CDCommand base class, it enables you to set the SELECT MESSAGE parameters. When you call the **Execute()** method, the command is executed and the resulting message text is stored in the internal CDMessage object.
CDStopCmd	Derived from the CDCommand base class, it enables you to set the STOP parameter. When you call the **Execute()** method, the command is executed.

CDSubmitCmd	Used for submitting a Process object for execution on a node. It enables you to set the options of the SUBMIT command and then execute the command on a node. When you call the **Execute()** method, a CDProcess object is dynamically created and attached to the connection assigned to execute the command. The following example demonstrates the CDSubmitCmd class:
	`.`
	`.`
	`.`
	`CDSubmitCmd cmd;`
	`cmd.SetFile ("myproc.cdp");`
	`CDProcess proc = node.Execute(cmd);`
	`proc.WaitForCompletion();`
	`.`
	`.`
	`.`

CDTraceOnCmd	Derived from the CDCommand base class, it enables you to set and retrieve trace options from the Connect:Direct node. The TraceOnCmd class handles all the options available from the TRACEON command. The **Execute()** method returns a CDTrace object that contains the current trace state.
CDTraceOffCmd	Derived from the CDCommand base class, it enables you to clear trace options from the Connect:Direct node. The CDTraceOffCmd class handles all of the options available from the TRACEOFF command. You call methods to clear the desired trace parameters and then call the **Execute()** method. The **Execute()** method returns a CDTrace object that contains the current trace state.

Managing Exception Conditions

Connect:Direct generates Exception Group classes if an exception condition is encountered while a request is being processed. Following is an exception scenario where a message is pushed into the exception before the initial throw.

Function A calls Function B, and Function B calls Function C. Function C is a helper routine called by many routines so it does not include information specific to a task. Since the exception occurred in C, it throws the exception. A message describing the error is added and flagged as a *technical message*.

Function B traps the exception. A message describing the error is added and flagged as a *user message.* User messages are displayed in dialog boxes. For example, a user message reads: *Communication with the server has been lost.*

The CDMsgException class stores the messages as an array of strings. The messages are stored in a last-in first-out (LIFO) order because messages added later are more general as the exception moves up the call stack.

Following is a description of the Exception Group classes:

Class	Description
CDMsgException	The base exception class for all Connect:Direct exception objects. It provides a message stack for troubleshooting.
CDConnectionException	This exception is generated when communication with the node is lost or cannot be established.
CDCommandException	Generated when an object cannot be executed because parameters are invalid, including submitted Process containing errors.
CDLogonException	Generated if the Connect:Direct node rejects the user ID and password supplied in the logon attempt. You can respond to this exception by prompting the user for the correct logon information.

Managing Administrative Functions

The Connect:Direct Helper Group classes are provided for convenience. They represent common functionality that can be used for consistency.

Class	Description
CDLogonDlg	The Connect:Direct common logon dialog box enables you to write your own logon applications. The CDLogon dialog box enables you to change the node, the user ID and password to connect to the Connect:Direct node as well as enable the **Remember Password** check box, click the **Configure** button to save new server logon information and change the title. Below are the components of the CDLogonDlg class: **Node**—Specifies the Connect:Direct node to which the user wants to logon. userid—Specifies the user ID for the Connect:Direct node. **Password**—Specifies the password defined for the user ID. **Remember Password**—Specifies whether the user wants the password to persist after the user logs off. If the check box is enabled, the password is retrieved to set the password field of the dialog box when the logon dialog is displayed. This prevents the user from having to re-type the password information for the session. Enabling the check box also specifies whether or not to write the password information as nonvolatile data. Nonvolatile keys persist after the user logs off. If the user does not enable the Remember Password check box, the password only persists until the user logs off. The Connect:Direct Logon dialog box does not perform the logon. It captures the entries and returns them to the calling program. Normally, the programmer creates a CDLogon dialog box, sets the parameters, and calls the **DoModal()** function to display and run the dialog box. If the user clicks the **OK** button, then the CDLogonDlg class returns IDOK and a logon is attempted using the supplied connection information. If the user clicks the **Cancel** button, the CDLogonDlg class returns IDCANCEL and the logon is cancelled. After a user successfully logs on to the Connect:Direct node, the connection information is written to the Registry under the HKEY_CURRENT_USER key.
CDExceptionDlg	Displays the exception dialog box. The dialog box displays the information in the exception object.
CDThread	Coordinates the clean termination of threads and provides a thread class that can unblock object.
CDBeginThread	Creates a worker thread for use with API objects.
Return Values	A pointer to the newly created thread object.

Creating a Thread Example

The following example illustrates how to create a thread:

```
void SomeFunc()
{
  CDThread* pThread = CDBeginThread(ThreadFunc);
}

void ThreadFunc(LPARAM lParam)
{
   CSomeCmd cmd(...);
   CDProcess proc = cmd.Execute(...);
   DWORD dwId = proc.GetId();
   SetDlgItemInt(IDC_SOMECONTROL, (int)dwId);
}
```

Terminating a Thread

In the preceding sample code, the only blocking that takes place is in the **Execute()** function. **Execute()** blocks until the Process information returns from the server. To terminate the thread without waiting, call **CDThread::Exit**, which signals any blocking CD objects in the thread to stop blocking and throw a thread exit exception. In the previous example, if **CDThread::Exit** is called, an exception is thrown, and no return object is returned from the **Execute()** function.

Note: It is not possible for one thread to throw an exception in another. **CDThread::Exit** sets flags in the CDThread object that other CD objects use.

When **CDThread::Exit** is called, **CDThread::IsExiting** returns TRUE. You can use this method in loops to determine when to exit because CD objects only throw the exception when they are blocking.

Caution: Do not call the Win32 TerminateThread. TerminateThread does not give the thread a chance to shut down gracefully. Calling TerminateThread can corrupt the state of the CD objects. CD objects use critical sections and other resources that must be managed carefully.

Catching the Exception

It is not necessary to catch the CDThreadDeath exception. If not caught, the exception unwinds the stack, destroying all objects on the stack, and the CDThread object itself handles the exception. To provide clean-up for heap allocated items, the exception can be caught. Rethrowing the exception is not required.

Understanding Multithreaded Access and Blocking

Because the Connect:Direct C++ Class API uses multiple threads, the API objects are thread safe. The API objects provide efficient blocking for use in multithreaded programs.

Using Objects on the Stack

C++ programs that make good use of exceptions move as much data from the heap to the stack as possible. This ensures that destructors run and memory is released when an exception occurs. It also reduces the complexity of the program by eliminating many pointers, reducing the chances of memory leaks, and letting the compiler ensure that objects are valid (as opposed to pointers that could be NULL or bad).

To ensure objects are used on the stack efficiently, most CD objects store their data externally. The following example is of an iterator object that holds 500 statistics records:

When the iterator is created, an iterator data object is also created to hold the records. The data object also has a reference count that indicates how many objects are using the data. When an object is copied, the new object (the copy) is linked to the data and the reference count of the data object is incremented. There are still only 500 records (not 1000), and the reference count is now 2.

When connected objects are destroyed, they decrement the reference count in the data object. When the reference count reaches 0, the data object is also destroyed. The following figure provides an example of the efficiency possible when shared data is copied:

```
1. void Func()
2. (
3.    Iterator itFinal = CreateIterator();
4. }
5.
6. Iterator CreateIterator()
7. {
8.    CSomeCmd cmd(...);
9.    Iterator itLocal = node.Execute(cmd);
10.    return itLocal;
11. }
```

On line 3 the sample code calls the **CreateIterator()** function. The **CreateIterator()** function returns an iterator, called *itLocal*. This iterator is created on line 9 and returned on line 10.

At line 11 the C++ compiler creates a temporary copy of *itLocal* before destroying it. As part of the copy, the iterator data reference count is incremented to 2. When *itLocal* is destroyed, the reference count drops to 1 so that the records are not deleted.

Next, the C++ compiler constructs *itLocal* on line 3 by passing the temporary to its copy constructor. The reference count is again incremented to 2 because both iterators are pointing to it. The temporary is then destroyed, reducing the reference count to 1.

The result is that an unlimited number of records are passed to the stack with little more than the copying of two pointers and some reference counting.

Chapter 5

Applying the ActiveX Control Interface

This chapter describes how to use the Connect:Direct ActiveX control interface. The CDSubmit and CDStatistics controls provide mechanisms to submit Processes to the server and display statistics from the statistics database.

The Connect:Direct ActiveX control interface provides CDSubmit and CDStatistics controls. Refer to the VBSTAT.VBP and VBSUBMIT2.VBP files in the Samples subdirectory for a sample Visual Basic application that uses the ActiveX controls included in this chapter.

Submitting Processes

The Connect:Direct CDSubmit control is a command line control that submits Processes to the server. Because submitting a Process can be a lengthy procedure, the **Execute** command returns immediately. When a Process is submitted and the server responds, or a time-out occurs, the client is notified through the SubmitStatus event. Additionally, the client can request notification when the Process has completed on the server. Properties for the CDSubmit control follow:

Property	Description
Node=*nodename*	The name of the node that you want to connect to. The node name must be valid in the Windows system Registry.
User=*userid*	The user ID used to log on to the Connect:Direct node.
Password=*password*	The password used by the user ID to log on to the node.
Text=*text*	The text of the Process.

Methods

Use the following methods to submit a Process:

Method	Description
Execute(BOOL bWait)	Submits the Process to the server. An event is fired when the server responds to notify the client of the status of the submit. If bWait is TRUE, another event is fired when the Process completes on the server.
SetSymbolic(symbolic, value)	Sets the symbolic value for symbolic. Call for each symbolic in the Process.
ClearSymbolics	Clears all symbolics. Call before submitting a Process to clear the previous values.

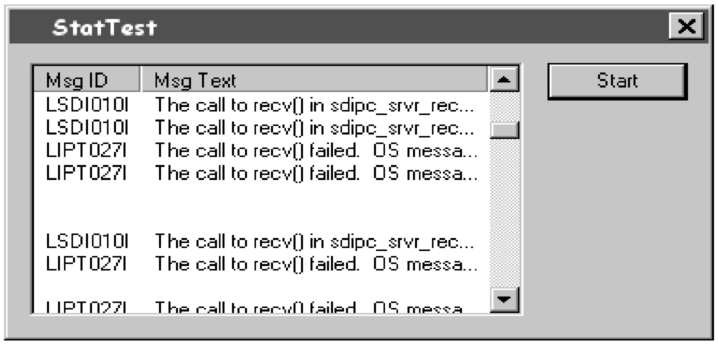
Events

The following events are activated by the CDSubmit control:

Method	Description
Submitted	Describes whether the Process is accepted by the server.
Completed	The ProcessComplete event is sent when the Process is no longer in the server's queue. Because more resources are required to wait on a Process, this event is only fired if requested in the call to Execute.
Error	The standard error event. Possible codes are: CTL_E_PERMISSIONDENIED—cannot log onto the node. CTL_E_DEVICEUNAVAILABLE—cannot connect to the node. CTL_E_OUTOFMEMORY—out of memory. CTL_E_ILLEGALFUNCTIONCALL—an unknown error. The error message describes the error.

Displaying Select Statistics Results

The CSDStatistics control is a multi-column list that displays SELECT STATISTICS command results. The CDStatistics control properties determine the node that you are connected to, logon information, and selection criteria. The following figure shows the CDStatistics control where only the message ID and message text are selected.



Msg ID	Msg Text
LSDI010I	The call to recv() in sdipc_srvr_rec...
LSDI010I	The call to recv() in sdipc_srvr_rec...
LIPT027I	The call to recv() failed. OS messa...
LIPT027I	The call to recv() failed. OS messa...
LSDI010I	The call to recv() in sdipc_srvr_rec...
LIPT027I	The call to recv() failed. OS messa...
LIPT027I	The call to recv() failed. OS messa

Properties

Properties for the CDStatistics control are listed below:

Property	Description
ColCount=*nnnnn*	Specifies the number of columns to display. The range for the ColCount value is 1–32,000.
Col=*nnnnn*	Specifies the current column. The range for the Col value is 1–32,000.
ColWidth=*nnnnn*	Specifies the width of the current column (Col) in pixels. The range for the ColWidth value is 0–32,000.
Header	Specifies the column header text for the current column. Provide text for the value or leave it blank.
Row=*nnnnn*...	Specifies the current row. If set to **0**, the current row is the header. The range for the Row value is 0–Infinity, where the number of rows is limited only by memory.
RowCount=*positive integer*	Specifies the number of rows in the list, not including the header. This field is read-only and is determined by the number of records returned by the server.
Node=*node name*	Specifies the name of the node to which you want to connect. The node name must be valid in the Windows NT system Registry.
User=***userid***	Specifies the user ID used to log on to the Connect:Direct node.
Password=*password*	Specifies the password defined to allow the user ID log on to the node.
Field	Specifies the statistics structure field the current column is displaying. Valid values are Process Name, Process Number, Condition Code, Feedback, MsgId, MsgText, MsgData, LogDateTime, StartDateTime, StopDateTime, Submitter, SNode, RecCat, and RecId.

Property	Description
ccode=(*operator*, *code*)	Selects statistics records based on the completion code operator and return code values associated with step termination. The condition code operator default is **eq**. You must specify the return code. Refer to *dfile=destination filename | (list)* on page 5-36 for valid operators and values.
dfile=*destination filename* | (*list*)	Searches all copy termination records (CAPR category, CTRC record ID) to find those with a destination file name matching the file name or list of file names specified. This parameter is not supported in a UNIX environment.
pname=*Process name* | *generic* | (*list*)	Selects Process statistics by Process name, a generic name, or a list of names. The name can be 1–8 alphanumeric characters long.
pnumber=*Process number* | (*list*)	Selects statistics by Process number or a list of Process numbers. Connect:Direct assigns the Process number when the Process is submitted.
reccat=caev | capr | (caev , capr)	Selects statistics based on whether the record category is related to events or to a Connect:Direct Process. The default for this keyword depends on the other search criteria specified. If you specify Process characteristics, such as Process name, Process number, or Submitter, the default is **capr**. If you perform a general search using **startt** or **stopt**, the default is **caev** and **capr**. **caev** specifies that the retrieved statistics file records include those related to Connect:Direct events, such as a Connect:Direct shutdown. capr specifies that the retrieved statistics file records include those related to one or more Connect:Direct Processes.
rnode=*remote node name* | *generic* | (*list*)	Selects statistics file records by remote node name, a generic node name, or a list of node names. The range for the remote node name is 1–16 alphanumeric characters long.
sfile=*filename* | (*list*)	Searches all **copy** Process Termination records (CAPR category, CTRC record ID) to find those with a source file name matching the name or list of names you specify.

Property	Description
startt=([*date* | *day*] [, *time*])	selects statistics starting with records logged since the specified date, day, or time. The date, day, and time are positional parameters. If you do not specify a date or day, type a comma before the time. *date* Specifies the day (dd), month (mm), and year (yy), which you can code as mm/dd/yyyy or mm-dd-yyyy. If you only specify date, the time defaults to 00:00:00. The current date is the default. *day* Specifies the day of the week. Values are today, yesterday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday. If you specify a day of the week, Connect:Direct uses the previous matching day. *time* Specifies the time of day coded as hh:mm:ss[am | pm] where hh is hours, mm is minutes, and ss is seconds. You can specify the hour in either 12- or 24-hour format. If you use the 12-hour format, then you must specify am or pm. The default format is the 24-hour format. The default value is 00:00:00, which indicates midnight. If you specify only the day value, the time defaults to 00:00:00.
stopt=([*date* | *day*] [, *time*])	Retrieves statistics including records logged up to and including the specified date, day, or time. The date, day, and time are positional parameters. If you do not specify a date or a day, type a comma before the time. *date* specifies the day (dd), month (mm), and year (yy), which you can code as mm/dd/yyyy or mm-dd-yyyy. If you only specify date, the time defaults to 00:00:00. The current date is the default. *day* specifies the day of the week. Values are today, yesterday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday. If you specify a day of the week, Connect:Direct uses the previous matching day. *time* specifies the time of day coded as hh:mm:ss[am | pm] where hh is hours, mm is minutes, and ss is seconds. You can specify the hour in either 12- or 24-hour format. If you use the 12-hour format, then you must specify am or pm. The default is the 24-hour format. The default value is 00:00:00, which indicates midnight. If you specify only the day value, the time defaults to 00:00:00.

Property	Description
submitter=(*node name*, *userid*) | *generic* | (*list*)	Selects statistics by the node name and user ID of the Process owner (submitter). You can also specify a generic name and user ID or a list of names and user IDs. The maximum combined length, including the node name and user ID, is 66 characters. Valid completion code operators for the ccode property are listed below: eq | = | ==Equal (default) ge | >= | =>Greater than or equal gt | >Greater than le | <= | =<Less than or equal lt | <Less than ne | !=Not equal Valid completion codes for the ccode property are listed below: 0—Successful execution of the Process. 4— A warning-level error was encountered. The statement probably completed normally, but verify the execution results. 8—An error occurred during Process execution. 16—A severe error occurred during Process execution.
recids=*record id* | (*list*)	Specifies selection by record ID or a list of record IDs. This parameter identifies particular types of statistics records, such as a copy termination records or initialization event records. Following is a list of the record IDs: APCK—Asset protection check AUPR—Authorization file processing CHGP—Change Process command issued COAC—Communication activated CMLT—CMGR listen thread terminated CRHT—Connect:Direct copyright CSTP—Child Process stopped CTRC—Copy control record written CTRM—Child Process terminated CUKN—Child process unknown status CXIT—Child process exited DELP—Delete Process command issued FLSP—Flush Process command issued FMRV—Formatted Header (FMH) received FMSD—Formatted Header (FMH) sent GPRC—Get Process issued

Property	Description
recids (continued)	IFED—If statement ended IPPR—Initialization parameter processing LIEX—License has expired LIOK—Listen okay LSST—The record ID of a step on the local node LWEX—License will expire in 14 days NAUH—Node Authorization check issued NMOP—Network map file opened NMPR—The network map is updated through Browser, Control Center, or KQV Interface. NUIC—Connect:Direct initialization complete NUIS—Connect:Direct start initialization NUT1—Connect:Direct phase one termination complete status NUT1—Connect:Direct phase two termination complete status NUTC—Connect:Direct termination complete NUTR—Connect:Direct termination requested PERR—Process error was detected PFLS—Process was flushed PMED—Process Manager ended PMIP—Process Manager Initprocs thread initialized PMMX—Process Manager Max Age thread initialized PMRC—Process Manager release cell thread initialized PMST—Process Manager started PPER—Pipe error PRED—Process ended PRIN—Process interrupted PSAV—Process was saved PSED—Process step was detected PSTR—Process has started QCEX—A Process moved from another queue to the EXEC queue QCHO—A Process moved from another queue to the HOLD queue QCWA—A Process moved from another queue to the WAIT queue QCTI—A Process moved from another queue to the TIMER queue QCHO—A Process moved from another queue to the HOLD queue RJED—Run Job command completed RNCF—Remote Connect:Direct server call failed RSST—The record ID of a step on the remote node RTED—Run Task command completed SBED—Submit complete SELP—Select Process command issued

Property	Description
recids (continued)	SELS—Select Statistics command issued SEND—Session end issued SERR—System error SFSZ—Size of the file submitted SHUD—Connect:Direct shutdown SIGC—System error SMED—Session Manager ended SMST—Session Manager started SNMP—SNMP SSTR—Session start issued STOP—Stop Connect:Direct command issued SUBP—Submit command issued TCPI—TCP started TRAC—Trace command issued TZDI—Time zone of the local node represented as the difference in seconds between the time at the local node and the Coordinated Universal Time UNKN—Unknown command issued USEC—User Security check issued xxxx—Record types identified by the first four characters of the message ID

Methods

The CDStatistics control provides the following methods:

Method	Description
BOOL Execute()	Executes the SELECT STATISTICS command and stores the returned records in the control. If the control was already retrieving records, the previous command is stopped and the old records are removed from the control.
Clear	Clears the existing records from the display. The Clear method does not stop retrieval.

Events

The following events are controlled by CDStatistics.

Method	Description
Complete	Sent after all records are retrieved.
Error	The standard error event. Possible codes are: CTL_E_PERMISSIONDENIED—cannot log onto the node. CTL_E_DEVICEUNAVALIABLE—cannot connect to the node. CTL_E_OUTOFMEMORY—out of memory. CTL_E_ILLEGALFUNCTIONCALL—an unknown error.

Chapter 6

Applying the Automation Servers

The Connect:Direct Automation Servers provide an automation wrapper around the Connect:Direct SDK C++ classes. They provide direct automation support for languages like Visual Basic. The following section provides a reference for the automation objects. Refer to *Creating Node Objects* on page 48 for information on applying the automation objects.

Creating Virtual Servers Using the Node Factory

The node factory creates node objects, which act as virtual servers. They represent a Connect:Direct server (node). The Automation Server Node Factory provides the following properties:

Property	**Description**
Node Name	Specifies the name of the node to connect to. The node name is set using the Connect:Direct Client Connection Utility.
Userid	Specifies the user ID to use when connecting to the node.
Password	Specifies the password to use for the user ID to connect to the node.

The Connect:Direct Automation Server Node provides the following methods:

Method	**Description**
SelectStats(criteria)	The criteria specifies the complete SELECT STATISTICS string.
SelectProc(criteria)	The criteria specifies the complete SELECT PROCESS string.
Submit(text)	The text specifies the Process to SUBMIT.

Applying the Node Object

The node object represents the connection to the Connect:Direct node. The node acts as the virtual server. The Connect:Direct Automation Server Node provides the following properties:

Property	Description
Name	Specifies the node name used while connecting. It is read only.
Userid	Specifies the user ID used while connecting. It is read only.

The Connect:Direct Automation Server Node provides the following methods:

Method	Description
SelectStats(criteria)	The *criteria* specifies the complete SELECT STATISTICS string.
SelectProc(criteria)	The *criteria* specifies the complete SELECT PROCESS string.

Identifying Active Processes

The Process object represents a Process running on the node. The records are returned as Process objects, stored in a ProcCollection container. The Connect:Direct Automation Server Process object provides the following properties:

Property	Type	Description
ProcessName	String	Specifies the Process name.
ProcessNumber	Long	Specifies the Process number assigned by Connect:Direct when the Process is placed in the TCQ.
ConditionCode	Long	Specifies the return code.
Feedback	Long	Provides additional return code information.
MsgId	String	Provides the message identifier field.
MsgText	String	Provides the message text field.
MsgData	String	Specifies the message substitution fields.
LogDateTime	Date	Provides the logged time stamp.
SchedDateTime	Date	Specifies the date and time the Process is scheduled to be submitted.
SubmitNode	String	Specifies the node name from which the Process was submitted.
Submitter	String	Specifies the user ID of the person submitting the Process.

Property	Type	Description
PNode	String	Specifies the primary or controlling node in the Process.
SNode	String	Specifies the secondary or partner node in the Process.
Status	String	Specifies the status of the Process in the queue.
Retain	String	Specifiers whether the Process is to be retained in the TCQ for future submission.
Hold	String	Specifies the TCQ hold status of the Process.
Class	Long	Specifies the session class on which the Process is executing.
Priority	Long	Specifies the TCQ selection priority of the Process.
ExecPriority	Long	Specifies the operating system execution priority of the Process.
Queue	String	Specifies the logical queue where the Process is currently located (Execution, Hold, Wait, or Timer).
Step Name	String	Specifies the currently executing step of the Process.
LocalNode	String	Specifies whether the primary or secondary node is the local node and has primary control.
FromNode	String	Specifies whether the primary or secondary node is the source node in a **copy**.
SimpleCompress	Boolean	Specifies to perform repetitive character compression.
ExtendedCompression	Boolean	Specifies to perform repetitive string compression.
Checkpoint	Boolean	Specifies the use of checkpointing in a copy step.
Restart	Boolean	Specifies whether the Process is restarted.
SourceFile	String	Specifies the name of the source file.
TotalBytes	Long	Specifies the number of data bytes read or written.
TotalRecs	Long	Specifies the number of data records read or written.
SentBytes	Long	Specifies the number of data bytes sent.
Sent RUs	Long	Specifies the number of RU bytes sent.
DestFile	String	Specifies the name of the destination file.

Identifying Statistic Records

The Statistic object represents the records in the statistics database. They are returned from a SELECT STATISTICS query. The Connect:Direct Automation Server Statistic object provides the following properties:

Property	Data Type	Description
ProcessName	String	Specifies the Process name.
ProcessNumber	Long	Specifies the Process number assigned by Connect:Direct when the Process is placed in the TCQ.
Feedback	Long	Provides additional return code information.
MsgId	String	Provides the message identifier field.
MsgText	String	Provides the message text field.
MsgData	String	Specifies the message substitution fields.
LogDateTime	Date	Provides the logged time stamp.
StartDateTime	Date	Specifies the start time stamp
StopDateTime	Date	Specifies the stop time stamp.
Submitter	String	Specifies the submitter’s user ID.
SNode	String	Specifies the secondary node name.
RecCat	String	Specifies the record category field.
RecId	String	Specifies the record identifier tag field.

Property	Data Type	Description
GetAuditField	String	Returns the audit field value. The **GetAuditField()** function supports the following audit information field names: ◆ Bytes Sent/Received ◆ Checkpoint ◆ Class ◆ Destination Disposition #1 ◆ Destination Disposition #2 ◆ Destination Disposition #3 ◆ Destination File ◆ Execution ◆ Extended Compression ◆ From Node ◆ Function ◆ Hold ◆ Local Node ◆ Priority ◆ Queue ◆ Remote Node ◆ Restart ◆ Retain ◆ RUs Sent/Received ◆ Scheduled Date/Time ◆ Source Disposition #1 ◆ Source Disposition #2 ◆ Source Disposition #3 ◆ Source File ◆ Standard Compression ◆ Status ◆ Step Name ◆ Submit Date/Time ◆ Total Bytes ◆ Total Records

Creating Node Objects, Select Processes, and Select Statistics Using Automation Objects

This section provides instructions on using the node factory and nodes, selecting statistics, and selecting Processes. The Connect:Direct automation objects use *late binding,* so you must dimension your variables as type Object.

Creating Node Objects

The Connect:Direct node factory creates node objects. These node objects serve as virtual servers and represent a connection to a Connect:Direct server (node).

To obtain a connection (and therefore a node), you must use the node factory. Create the node factory using the ProgID *CD.NodeFactory:*

```
Dim factory as Object
Set factory = CreateObject ("CD.NodeFactory")
```

To determine the node that you want to connect to, set the properties of the factory object. Next, call *CreateNode* to connect to the node. If the connection is successful, a node object returns. Otherwise, an error is thrown indicating the cause of the problem.

```
factory.NodeName = "CD.Node1"
factory.UserId = "user1"
factory.Password = "password"

Dim node as Object
Set node = factory.CreateNode()
```

The node name refers to the name used by the Client Connection Utility. You must set up the nodes that you want to connect to using the Client Connection Utility prior to using the Connect:Direct SDK.

Using Nodes

The node object represents the connection to a Connect:Direct node. Using the node enables you to select statistics or Processes.

Selecting Processes

To select Processes, you must first format a select Process command and pass it to the SelectProc method. The records return as Process objects and are stored in the ProcCollection container. Because a background thread populates the collection, it is returned to the caller before it is completely filled. Therefore, the only access method available is using the *For Each* construct.

Note: The usual Count property is not available because the count is not known until all the records are returned.

```
Dim procs as Object     ; the process collection
Dim proc as Object     ; each process record

Set procs = node.SelectProc ("SELECT PROCESS ")
For Each proc in procs
     Debug.Print proc.ProcessName
Next proc
```

Selecting Statistics

To select statistics records, you must format a select statistics command and pass it on to the SelectStats method of the node. The records return as Statistic objects stored in a StatCollection container. Because a background thread populates the collection, it returns to the caller before it is completely filled. Therefore, the only access method available is using the *For Each* construct.

Note: The usual Count property is not available because the count is not known until all records are returned.

```
Dim stats as object     ; the Statistics collection
Dim stat as Object     ; each statistic record

Set stats = node.SelectStats ("SELECT STATISTICS")
For Each stat in stats
    Debug.Print stat.RecId
Next stat
```

Because the server can send records slowly, the interface can be jerky while reading records. Because records are read using a background thread, it useful to select the statistics before time-consuming tasks like constructing windows. This method enables the server to send records in the background.

Understanding Errors

The automation classes use the standard Visual Basic error-handling mechanism. When an error is raised in an automation object, no real value is returned from the function. For example, if an error is raised in the node factory example in the *Creating Node Objects* on page 6-48, *node* does not have a value (it has the default value of *nothing*) because CreateNode has not returned anything.

When the Connect:Direct automation objects raise an error, they set the error number to a Connect:Direct SDK error value and store a description in the error text.

Chapter 7

Enhance Security and Automate File Opening with User Exits

You can customize Connect:Direct operations with user exits. User exits are user-defined dynamic link libraries (DLLs) that are loaded and called when the user exit is enabled through an initialization parameter. Two user exits are provided: one for enhanced security and one for automated file opening.

Applying Enhanced Security

You can implement enhanced security with passtickets. A passticket is a one-time password generated on the primary node and passed to the secondary node within 10 minutes, where it is validated before further processing is performed.

Applying Passticket Support

Connect:Direct passticket support is implemented as a user exit called from the Connect:Direct session manager during Process execution. To enable the security exit, change the value of the **security.exit** initialization parameter to the name or path name of the security exit DLL.

See Appendix B, *Changing Connect:Direct for Windows Settings,* in *Connect:Direct for Windows System Guide* for a description of the **security.exit** parameter. If the DLL is not in the search path of the server, then you must specify the fully qualified file name of the DLL.

The user's security exit must contain the **GeneratePassticket()** and **ValidatePassticket()** functions. The parameters for these functions are defined in the userexit.h header file. The userexit.h header file is in the Connect:Direct samples directory. If the security exit cannot be found or loaded, or if the addresses of the two required functions cannot be resolved successfully, an error message is generated and Process execution terminates.

- ✦ The passticket is only valid for 10 minutes after it is generated. As a result, the system clocks on the two nodes should be synchronized.
- ✦ When generating passtickets, Connect:Direct for Windows fills in the GENMSG_T structure fields and passes the structure to the security exit. The security exit should generate the

passticket, fill in the GENMSG_REPLY_T structure fields, and return an appropriate return code to Connect:Direct.

- When validating a passticket, Connect:Direct for Windows fills in the VALMSG_T structure fields and passes the structure to the security exit. The security exit validates the passticket, fills in the VALMSG_REPLY_T structure fields, and returns an appropriate return code to Connect:Direct. If the passticket is successfully validated, Connect:Direct for Windows continues as if the Process is using a remote user proxy.

Security Exit Structures

Following is a list of the security exit structures:

- GENERATE_MSG—Sends a message to the local node to allow the security exit to determine user ID and security token (passticket) to use for remote node authentication. The GENERATE_MSG contains:
  - Submitter ID
  - Local node ID and password
  - Remote node ID and password
  - Local node name
  - Remote node name
- GENERATE_MSG_REPLY—The user exit **GeneratePassticket()** function fills the GENERATE_MSG_REPLY structure. The GENERATE_MSG_REPLY contains:
  - Status value of GOOD_RC (0) for success, or ERROR_RC (8) for failure.
  - Status text message. If the status value is failure, then status text message is included in the error message.
  - ID to be used for security context on the remote node. This may or may not be the same ID that was passed in.
  - Passticket to use in conjunction with the ID for security on the remote node.
- VALIDATE_MSG—The message sent to the remote node to allow the security exit to validate the user ID and passticket. The VALIDATE_MSG contains:
  - Submitter ID
  - Local node ID and password
  - Remote node ID and password
  - Local node name
  - Remote node name
  - ID to be used for security checking
  - Passticket generated on the local node
- VALIDATE_MSG_REPLY—The user **ValidatePassticket0** function fills the VALIDATE_MSG_REPLY structure. The VALIDATE_MSG contains:
  - GOOD_RC (0) if the reply was a success or ERROR_RC (8) for failure.

- Status text message. If the status value is failure, the status text message is included in the error message.
- ID to be used for security context the remote node side. This value may or may not be the same ID as in the generate message.
- Passticket to use in conjunction with ID for security on the remote node.

Accessing Sample Code

The following header file and sample code files for passticket implementation are copied to *X*:*installation directory*\Samples during the installation.

- USEREXIT.H—Contains defined constants used for passtickets, the structures that are passed to the passticket functions, and the function prototypes.
- USERSKEL.C—Consists of the **GeneratePassticket()** and **ValidatePassticket()** functions. The **GeneratePassticket()** function generates the passticket nnnnnnn, fills in the structure, and returns a valid return code. The **ValidatePassticket()** function returns a good return code indicating that the passticket passed in is valid. There is no real checking done in this routine.
- USERSAMP.C—Demonstrates a full implementation of passticket support. The **GeneratePassticket()** and **ValidatePassticket()** functions call the **Passtk()** function which performs the actual generation, or validation of the passticket.

The sample user exit can be compiled and linked into a DLL using Microsoft Visual C++™. The exit was tested using Microsoft Visual C++ version 4.2.

Applying Automated File Opening

You can override the values specified in the COPY statement with the file open exit feature. The file open exit is an initialization parameter (**file.exit**) that you can set to point to a user-written DLL. You can customize Connect:Direct COPY operations by defining values in the file open exit DLL that override the COPY statement parameters.

Applying the File Open Exit

Connect:Direct file open support is implemented as a user exit called from the Connect:Direct session manager during Connect:Direct COPY statement execution. To enable the file open exit, change the value of the **file.exit** initialization parameter to the name or path name of the file open exit DLL. Refer to Appendix B, *Changing Connect:Direct Windows Settings,* in the *Connect:Direct for Windows System Guide* for a description of the **file.exit** parameter. If the DLL is not in the search path of the server, then you must specify the fully qualified file name of the DLL.

The user's file open exit must contain the **FileOpen(**) function. The parameters for this function are **File_Open** and **File_Open_Reply**. These parameters are pointers to corresponding structures in the *userexit.h* header file. The *userexit.h* header file is in the Connect:Direct samples directory.

File Open Exit Structures

The file open exit contains the following two types of structures:

FILE_OPEN

The FILE_OPEN structure contains the information that implements the file open user exit. The FILE_OPEN structure contains the following components:

✦ **int oflag—**Flags that Connect:Direct uses to open the file.

✦ **int srcdstflag—**Specifies whether the file is a *source* file (the file to read) or a *destination* file (the file to write to).

✦ **char user_name[MAX_USER_NAME]—**Specifies the name of the user that submitted the Process.

✦ **COPY_T copy_ctl**—Points to the Connect:Direct Copy Control Block data structure that contains information concerning the COPY operation about to be performed.

✦ **COPY_SYSOPTS_T cp_sysopts**—Points to the Sysopts data structure that contains a representation of all of the COPY operation sysopts that Connect:Direct supports. Refer to the Connect:Direct Process Web site for more information about COPY sysopts.

FILE_OPEN_REPLY

The FILE_OPEN_REPLY structure contains the information that specifies whether the file exit operation succeeded. The FILE_OPEN structure contains the following components:

✦ **HANDLE hFile—**Contains a valid file handle if the file was opened successfully.

✦ **char filename[MAX_FILE_NAME_LEN]—**Contains the actual name of the file opened by the file open exit.

Accessing Sample Code

The following header file and sample code files for file open exit implementation are copied to *X*:*installation directory*\Samples during Connect:Direct for Windows installation.

✦ userexit.h

✦ FileOpenDLL.CPP

Appendix A

Structure Types

This appendix contains a list of the common C and C++ Class interface structures, constants, and their descriptions. All of the common C and C++ Class API structures are contained within the CONNDIR.H header file.

NETMAP_DESC_STRUCT Structure

The NETMAP_DESC_STRUCT structure contains the Netmap Node Description information. This structure is used for retrieving and setting the Netmap Node Description information. The following figure represents the NETMAP_DESC_STRUCT structure:

```
struct Netmap_Desc_Struct
{
    TCHAR Name[MAX_NODE_NAME_LEN+1];
    TCHAR ContactPhone[MAX_PHONE_NUMBER+1];
    TCHAR ContactName[MAX_CONTACT_NAME+1];
    TCHAR Description[MAX_DESCRIPTION+1];
};
typedef struct Netmap_Desc_Struct NETMAP_DESC_STRUCT;
```

Members

The NETMAP_DESC_STRUCT structure contains the following members:

Member	**Description**
Name [MAX_NODE_NAME_LEN+1]	Specifies the node name.
ContactPhone [MAX_PHONE_NUMBER+1]	Specifies the phone number of the person responsible for this node.
ContactName [MAX_CONTACT_NAME+1]	Specifies the name of the person responsible for this node.
Description [MAX_DESCRIPTION+1]	Specifies the node description information.

USER_STRUCT Structure

The USER_STRUCT structure contains the User Functional Authority information. This structure is used for retrieving and setting the User Functional Authority information.

The following figure represents the USER_STRUCT structure:

```
struct User_Struct
{   TCHAR Name [MAX_OBJECT_NAME+1];
    TCHAR UpdateNetmap;
    TCHAR UpdateUser;
    TCHAR UpdateProxy;
    TCHAR ChangeProcess;
    TCHAR DeleteProcess;
    TCHAR SelectProcess;
    TCHAR SubmitProcess;
    TCHAR SelectStats;
    TCHAR SecureRead;
    TCHAR SecureWrite;
    TCHAR Stop;
    TCHAR Trace;
    TCHAR SelectNetmap;
    TCHAR SelectMessage;
    TCHAR Refresh;
    TCHAR ProcessCopy;
    TCHAR ProcessRunJob;
    TCHAR ProcessRunTask;
    TCHAR ProcessSubmit;
    TCHAR InheritRights;
    TCHAR TrusteeAssign;
    TCHAR UpdateACL;
    TCHAR FileAttributes;
    TCHAR SNodeId;
    TCHAR ExecutionPriority;
    TCHAR ProcessSend;
    TCHAR ProcessReceive;
    TCHAR UpdateTranslation;
    TCHAR DownloadDirectory[MAX_DIRECTORY_NAME+1];
    TCHAR UploadDirectory[MAX_DIRECTORY_NAME+1];
    TCHAR ProcessDirectory[MAX_DIRECTORY_NAME+1];
    TCHAR ProgramDirectory[MAX_DIRECTORY_NAME+1];
};
typedef struct User_Struct USER_STRUCT;
```

Members

The USER_STRUCT structure contains the following members:

Member	Description
Name [MAX_OBJECT_NAME+1]	Specifies the user name for this functional authority record.
SpecifiesUpdateNetmap	Specifies the permission to update the network map.

Member	Description
UpdateUser	Specifies the permission to update other user functional authority.
UpdateProxy	Specifies the permission to update proxy user information.
ChangeProcess	Specifies the permission to issue CHANGE PROCESS.
DeleteProcess	Specifies the permission to issue DELETE PROCESS.
SelectProcess	Specifies the permission to issue SELECT PROCESS.
SubmitProcess	Specifies the permission to issue SUBMIT PROCESS.
SelectStats	Specifies the permission to issue SELECT STATISTICS.
SecureRead	Specifies the permission to read Secure+ network map fields.
SecureWrite	Specifies the permission to modify Secure+ network map fields.
Stop	Specifies the permission to issue the STOP Connect:Direct server command.
Trace	Specifies the permission to start and stop Connect:Direct tracing.
SelectNetmap	Specifies the permission to get the network map objects from the Connect:Direct server.
SelectMessage	Specifies the permission to get Connect:Direct message information from the Connect:Direct server.
Refresh	Specifies the permission to execute the REFRESH INITPARMS commands.
ProcessCopy	Specifies the permission to issue a COPY command within a Process.
ProcessRunJob	Specifies the permission to issue a RUN JOB command within a Process.
ProcessRunTask	Specifies the permission to issue a RUN TASK command within a Process.
ProcessSubmit	Specifies the permission to issue a SUBMIT command within a Process.
Inherit Rights	Specifies the Inherit Rights flag.
TrusteeAssign	Specifies the Trustee Assign flag.
UpdateACL	Specifies the Update ACL flag.
FileAttributes	Specifies the File Attribute flag.
SNodeId	Specifies the Remote Node ID flag.

Member	Description
ExecutionPriority	Specifies the permission to change execution priority.
ProcessSend	Specifies the Process Send flag.
ProcessReceive	Specifies the Process Receive flag.
UpdateTranslation	Specifies the permission to update the translation table information.
DownloadDirectory [MAX_DIRECTORY_NAME+1]	Specifies the default download directory.
UploadDirectory [MAX_DIRECTORY_NAME+1]	Specifies the default upload directory.
ProcessDirectory [MAX_DIRECTORY_NAME+1]	Specifies the default Process file directory.
ProgramDirectory [MAX_DIRECTORY_NAME+1]	Specifies the default program file directory.

MESSAGE_STRUCT Structure

The MESSAGE_STRUCT structure contains the Connect:Direct message information. This structure is used for retrieving the Connect:Direct message information. It contains the unique message identifier.

The following figure represents the MESSAGE_STRUCT structure:

```
struct Message_Struct
{
    TCHAR MsgId[MAX_MESSAGE_ID+1];
    int ConditionCode;
    int Feedback;
    TCHAR MsgText[MAX_MESSAGE_TEXT+1];
    TCHAR MsgData[MAX_MESSAGE_DATA+1];
};
typedef struct Message_Struct MESSAGE_STRUCT;
```

Members

The MESSAGE_STRUCT structure contains the following members:

Member	Description
MsgId [MAX_MESSAGE_ID+1]	Specifies the message identifier that uniquely identifies this message.
ConditionCode	Specifies the return code accompanying the message.

Member	Description
Feedback	Specifies additional return code information.
MsgText [MAX_MESSAGE_TEXT+1]	Specifies the message text.
MsgData [MAX_MESSAGE_DATA+1]	Specifies the message substitution fields.

NETMAP_MODE_SNA Structure

The NETMAP_MODE_SNA structure contains the Netmap SNA Mode information. This structure is part of the NETMAP_MODE_STRUCT for SNA modes.

The following figure represents the NETMAP_MODE_SNA structure:

```
struct Netmap_Mode_Sna
{
    long lMaxRUSize;
    short MaxPacingSize;
    short MaxNetSessLimit;
};
typedef struct Netmap_Mode_Sna NETMAP_MODE_SNA;
```

Members

The NETMAP_MODE_SNA structure contains the following members

Member	Description
lMaxRUSize	Specifies the maximum RU size.
MaxPacingSize	Specifies the maximum pacing size.
MaxNetSessLimit	Specifies the maximum net session limit.

NETMAP_MODE_STRUCT Structure

The NETMAP_MODE_STRUCT structure contains the Netmap Mode information. This structure is used for retrieving and setting the Netmap Mode information.

The following figure represents the NETMAP_MODE_STRUCT structure:

```
struct Netmap_Mode_Struct
{
    TCHAR Name[MAX_OBJECT_NAME+1];
    BOOL bDetail;
    int Protocol;
    union
    {
    NETMAP_MODE_SNA    Sna;
    NETMAP_MODE_TCP    Tcp;
    }
    Type;
};
typedef struct Netmap_Mode_Struct NETMAP_MODE_STRUCT;
```

Members

The MODE_STRUCT structure contains the following members:

Member	Description
Name [MAX_OBJECT_NAME+1]	Specifies the mode name.
bDetail	Specifies the detail flag.
Protocol	Specifies the mode protocol.
Sna	MODE_SNA contains the SNA mode-specific fields.
Tcp	MODE_TCP contains the TCP/IP mode-specific fields.

NETMAP_MODE_TCP Structure

The NETMAP_MODE_TCP structure contains the Netmap TCP/IP Mode information. This structure is part of the NETMAP_MODE_STRUCT for TCP/IP modes.

The following figure represents the NETMAP_MODE_TCP structure:

```
struct Netmap_Mode_Tcp
{
    long lBufferSize;
    long lPacingSendCount;
    long lPacingSendDelay;
    char tcp_crc[4];
};
typedef struct  Netmap_Mode_Tcp NETMAP_MODE_TCP;
```

Members

The NETMAP_MODE_TCP structure contains the following members:

Member	Description
lBufferSize	Specifies the buffer size.
lPacingSendCount	Specifies the pacing send count.
lPacingSendDelay	Specifies the pacing send delay.
char tcp_crc[4]	Specifies if TCP CRC checking is on.

NETMAP_NODE_STRUCT Structure

The NETMAP_NODE_STRUCT structure contains the Netmap node information. This structure is used for retrieving and setting the Netmap node information.

The following figure represents the NETMAP_NODE_STRUCT structure:

```
struct Netmap_Node_Struct
{
    TCHAR Name[MAX_OBJECT_NAME_LEN+1];
    BOOL bDetail;
    int LongTermRetry;
    long lLongTermWait;
    int ShortTermRetry;
    long lShortTermWait;
    int MaxPNode;
    int MaxSNode;
    int DefaultClass;
    int RemoteOSType;
    TCHAR TcpModeName[MAX_OBJECT_NAME+1];
    TCHAR TcpAddress[MAX_TCP_ADDRESS+1];
    TCHAR SnaModeName[MAX_OBJECT_NAME+1];
    TCHAR SnaNetName[MAX_NET_NAME+1];
    TCHAR SnaPartnerName[MAX_PARTNER_NAME+1];
    TCHAR SnaTPName[MAX_TPNAME+1];
};
typedef struct Netmap_Node_Struct NETMAP_NODE_STRUCT;
```

Members

The NETMAP_NODE_STRUCT structure contains the following members:

Member	Description
Name [MAX_OBJECT_NAME_LEN+1]	Specifies the node name.
bDetail	Specifies detail included flag.
LongTermRetry	Specifies the long-term retry interval.
lLongTermWait	Specifies the long-term wait interval.
ShortTermRetry	Specifies the short-term retry interval.
lShortTermWait	Specifies the short-term wait interval.
MaxPNode	Specifies the maximum local nodes.
MaxSNode	Specifies the maximum remote nodes.
DefaultClass	Specifies the default class.
RemoteOSType	Specifies the node operating system type.
TcpModeName [MAX_OBJECT_NAME+1]	Specifies the TCP/IP communications mode name.
TcpAddress [MAX_TCP_ADDRESS+1]	Specifies the nodes TCP/IP address.
SnaModeName [MAX_OBJECT_NAME+1]	Specifies the SNA communications mode name.
SnaNetName [MAX_NET_NAME+1]	Specifies the SNA net name.
SnaPartnerName [MAX_PARTNER_NAME+1]	Specifies the partner name.
SnaTPName [MAX_TPNAME+1]	Specifies the TP name.

NETMAP_PATH_STRUCT Structure

The NETMAP_PATH_STRUCT structure contains the Netmap path information. This structure is used for retrieving and setting the Netmap path information.

The following figure represents the NETMAP_PATH_STRUCT structure:

```
struct  Netmap_Path_Struct
{  TCHAR  Name[MAX_OBJECT_NAME+1];
   BOOL   bDetail;
   int    Transport;
   int    Adapter;
   BYTE   Address[MAX_ADDRESS];
   char   CustomQLLC[MAX_CUSTOM_ADDRESS+1];
   int    Protocol;
   TCHAR  SnaProfileName[MAX_PROFILE_NAME+1];
   TCHAR  SnaLocalNetId[MAX_LOCALNETID+1];
   TCHAR  SnaPUName[MAX_PUNAME+1];
   TCHAR  SnaLUName[MAX_LUNAME+1];
   int    SnaLULocAddr;
   int    SnaLUSessLimit;
   int    TCPMaxTimeToWait;
   int    DialupHangon;
   char   DialupEntry[MAX_DIALUP_ENTRY+1];
   char   DialupUserid[MAX_OBJECT_NAME+1];
   char   DialupPassword[MAX_OBJECT_NAME+1];
   TCHAR  ModeName[MAX_OBJECT_NAME+1];
};
typedef struct Netmap_Path_Struct  NETMAP_PATH_STRUCT;
```

Members

The NETMAP_PATH_STRUCT structure contains the following members:

Member	Description
Name [MAX_OBJECT_NAME+1]	Specifies the path name.
bDetail	Specifies the detail flag.
Transport	Specifies the transport type.
Adapter	Specifies the adapter.
Address [MAX_ADDRESS]	Specifies the adapter address.
CustomQLLC[MAX_CUSTOM_ADDRESS+1]	Specifies the custom or QLLC adapter address.
Protocol	Specifies the protocol type.
SnaProfileName[MAX_PROFILE_NAME+1]	Specifies the SNA profile name.
SnaLocalNetId [MAX_LOCALNETID+1]	Specifies the SNA local net ID.
SnaPUName [MAX_PUNAME+1]	Specifies the SNA PU name.
SnaLUName [MAX_LUNAME+1]	Specifies the SNA LU name.
SnaLULocAddr	Specifies the SNA LU local address.
SnaLUSessLimit	Specifies the SNA LU session limit.

Member	Description
TCPMaxTimeToWait	Specifies TCP maximum time to wait.
DialupHangon	Specifies seconds to stay connected after dialup hangon completes.
DialupEntry[MAX_DIALUP_ENTRY+1]	Specifies dialup entry name.
DialupUserid[MAX_OBJECT_NAME+1]	Specifies dialup user ID.
DialupPassword[MAX_OBJECT_NAME+1]	Specifies dialup password.
ModeName [MAX_OBJECT_NAME+1]	Specifies the mode name used by this path.

PROCESS_STRUCT Structure

The PROCESS_STRUCT structure contains the Connect:Direct Process information. This structure is sent to the client from the Connect:Direct server upon accepting a Process for execution. It is also sent in response to a SELECT PROCESS command. It contains the Process name, Process number, and queue.

The following figure represents the PROCESS_STRUCT structure:

```
struct Process_Struct
{
    TCHAR ProcessName[MAX_PROCESS_NAME+1];
    DWORD ProcessNumber;
    int ConditionCode;
    int Feedback;
    TCHAR MsgId[MAX_MESSAGE_ID+1];
    TCHAR MsgText[MAX_MESSAGE_TEXT+1];
    TCHAR MsgData[MAX_MESSAGE_DATA+1];
    time_t LogDateTime;
    time_t SchedDateTime;
    TCHAR SubmitNode[17];
    TCHAR Submitter[65];
    TCHAR PNode[17];
    TCHAR SNode[17];
    TCHAR Status[3];
    TCHAR Retain;
    TCHAR Hold;
    int Class;
    int Priority;
    int ExecPriority;
    TCHAR Queue[5];
    TCHAR Function[6];
    TCHAR StepName[9];
    TCHAR LocalNode;
    TCHAR FromNode;
    BOOL bStandardCompression;
    BOOL bExtendedCompression;
    BOOL bCheckpoint;
    BOOL bRestart;
TCHAR SourceFile[MAX_FILENAME+1];
TCHAR SourceDisp1;
TCHAR SourceDisp2;
TCHAR SourceDisp3;
__int64 ByteCount;
__int64 RecordCount;
   __int64 XmitBytes;
long XmitRUs;
     TCHAR DestFile[MAX_FILENAME+1];
     TCHAR DestDisp1;
     TCHAR DestDisp2;
     TCHAR DestDisp3;
//SECURE_PLUS
    BOOL bSecurePlusEnabled;
    TCHAR EncAlgName[MAX_OBJECT_NAME];
    BOOL bSignature;
};
typedef struct Process_Struct PROCESS_STRUCT;
```

Members

The PROCESS_STRUCT structure contains the following members:

Member	**Description**
ProcessName [MAX_PROCESS_NAME+1]	Specifies the Process name.
ProcessNumber	Specifies the Process number.
ConditionCode	Specifies the return code.
Feedback	Specifies additional return code information.
MsgId [MAX_MESSAGE_ID+1]	Specifies the message identifier field.
MsgText [MAX_MESSAGE_TEXT+1]	Specifies the message text field.
MsgData [MAX_MESSAGE_DATA+1]	Specifies the message substitution data.
LogDateTime	Specifies the logged time stamp.
SchedDateTime	Specifies the scheduled time stamp.
SubmitNode [17]	Specifies the submitter’s node.
Submitter [65]	Specifies the submitter’s user name.
PNode [17]	Specifies the primary node.
SNode [17]	Specifies the secondary node.
Status [3]	Specifies the current status.
Retain	Specifies the retain flag.
Hold	Specifies the hold flag.
Class	Specifies the class.
Priority	Specifies the current priority.
ExecPriority	Specifies the current execution priority.
Queue [5]	Specifies the current queue that contains this Process.
Function[6]	Specifies the function executing in the Process.
StepName [9]	Specifies the current step name.
LocalNode	Specifies the local node flag.
FromNode	Specifies the from node flag.
bStandardCompression	Specifies the standard compression indicator.
bExtendedCompression	Specifies the extended compression indicator.
bCheckpoint	Specifies the checkpointing enabled indicator.

Member	Description
bRestart	Restart indicator.
SourceFile [MAX_FILENAME+1]	Specifies the source file name.
SourceDisp1	Specifies the source displacement 1.
SourceDisp2	Specifies the source displacement 2.
SourceDisp3	Specifies the source displacement 3.
ByteCount	Specifies the total byte count.
RecordCount	Specifies the total record count.
XmitBytes	Specifies the sent byte count.
XmitRUs	Specifies the sent RU count.
DestFile[MAX_FILENAME+1]	Specifies the destination file name.
DestDisp1	Specifies the destination displacement 1.
DestDisp2	Specifies the destination displacement 2.
DestDisp3	Specifies the destination displacement 3.
bSecurePlusEnabled	Specifies the Secure+ enabled flag.
EncAlgName[MAX_OBJECT_NAME]	Specifies the effective encryption algorithm.
bSignature	Specifies the effective signature setting.

NODE_STRUCT Structure

The NODE_STRUCT structure contains the Connect:Direct node information. This structure contains the Connect:Direct node name, the login information, operating system information, and protocol information. This information is stored in the Registry and is sent to the client after successfully logging on.

The following figure represents the NODE_STRUCT structure:

```
struct Node_Struct
{
    TCHAR Name[MAX_NODE_NAME_LEN+1];
    TCHAR CDName[MAX_NODE_NAME_LEN+1];
    TCHAR Server[MAX_OBJECT_NAME+1];
    long ApiVersion;
    long SecurePlusVersion;
    int CompLevel;
    int SelectedOSType;
    int OSType
    int SubType
    TCHAR Userid[MAX_OBJECT_NAME+1];
    TCHAR Password[MAX_OBJECT_NAME+1];
    BOOL bTemporary;
    BOOL bRememberPW;
    int Protocol
    TCHAR TcpAddress[MAX_TCP_ADDRESS+1]
};
typedef struct Node_Struct NODE_STRUCT;
```

Members

The NODE_STRUCT structure contains the following members

Member	**Description**
Name [MAX_NODE_NAME_LEN+1]	Specifies the Connect:Direct node alias name.
CDName [MAX_NODE_NAME_LEN+1]	Specifies the Connect:Direct node name.
Server [MAX_OBJECT_NAME+1]	Specifies the file server name.
ApiVersion	Specifies the API version.
SecurePlusVersion	Specifies the Secure+ version; value is 0 if Secure+ is not supported.
CompLevel	Specifies the KQV Communications Compatibility Level.
SelectedOSType	Specifies the user-selected operating system type.
OSType	Specifies the operating system type.
SubType	Specifies the sub type information.
Userid [MAX_OBJECT_NAME+1]	Specifies the user name.
Password [MAX_OBJECT_NAME+1]	Specifies the user-defined password.
bTemporary	Specifies to hold the user information temporary.
bRememberPW	Specifies to save the password in the Registry.
Protocol	Specifies the protocol type.

STATISTICS_STRUCT Structure

The STATISTICS_STRUCT structure contains the Connect:Direct statistics information for a Process. This structure is sent to the client as a result of a SELECT STATISTICS command.

The following figure represents the STATISTICS_STRUCT structure:

```
struct Statistic_Struct
{
    TCHAR ProcessName[MAX_PROCESS_NAME+1];
    DWORD ProcessNumber;
    int ConditionCode;
    int Feedback;
    TCHAR MsgId[MAX_MESSAGE_ID+1];
    TCHAR MsgText[MAX_MESSAGE_TEXT+1];
    TCHAR MsgData[MAX_MESSAGE_DATA+1];
    time_t LogDateTime;
    time_t StartDateTime;
    time_t StopDateTime;
    TCHAR Submitter[65];
    TCHAR SNode[17];
    TCHAR RecCat[5];
    TCHAR RecId[5];
};
typedef struct Statistic_Struct STATISTIC_STRUCT;
```

Members

The STATISTICS_STRUCT structure contains the following members:

Member	Description
ProcessName [MAX_PROCESS_NAME+1]	Specifies the Process name.
ProcessNumber	Specifies the Process number.
ConditionCode	Specifies the return code.
Feedback	Specifies the additional return code information.
MsgId [MAX_MESSAGE_ID+1]	Specifies the message identifier field.
MsgText [MAX_MESSAGE_TEXT+1]	Specifies the message text field.
MsgData [MAX_MESSAGE_DATA+1]	Specifies the message substitution data.
LogDateTime	Specifies the logged time stamp.
StartDateTime	Specifies the start time stamp.
StopDateTime	Specifies the stop time stamp.
Submitter [65]	Specifies the submitter’s user ID.

Member	Description
SNode [17]	Specifies the secondary node name.
RecCat [5]	Specifies the record category field.
RecId [5]	Specifies the record identifier tag field.

TRACE_STRUCT Structure

The TRACE_STRUCT structure contains the trace information. This structure is used for retrieving the trace information.

The following figure represents the TRACE_STRUCT structure:

```
struct Trace_Struct
{
    TCHAR cMainLevel;
    TCHAR cCommLevel;
    TCHAR cCMgrLevel;
    TCHAR cPMgrLevel;
    TCHAR cSMgrLevel;
    TCHAR cStatLevel;
    TCHAR szFilesize[MAX_FILENAME+1];
    long cbFilesize;
    BOOL bWrap;
    BOOL bPNode;
    BOOL bSNode;
    int PNums[4];
    TCHAR PNames[4] [MAX_PROCESS_NAME+1];
    TCHAR DestNodes[4] [17];
};
typedef struct Trace_Struct TRACE_STRUCT;
```

Members

The TRACE_STRUCT structure contains the following members:

Member	Description
cMainLevel	Specifies the MAIN trace level.
cCommLevel	Specifies the COMM trace level.
cCMgrLevel	Specifies the CMGR trace level.
cPMgrLevel	Specifies the PMGR trace level.
cSMgrLevel	Specifies the SMGR trace level.

Member	Description
cStatLevel	Specifies the STAT trace level.
szFilename[MAX_FILENAME+1]	Specifies the trace file name.
cbFilesize	Specifies the size of the trace file.
bWrap	Specifies whether to wrap when cbFile is reached.
bPNode	Specifies the PNODE trace flag.
bSNode	Specifies the SNode trace flag.
PNums[8]	Specifies an integer array of up to four Process numbers.
PNames[8] [MAX_PROCESS_NAME+1]	Specifies the string array of Process names.
DestNodes[8] [17]	Specifies the string array of destination node names.

TRANSLATE_STRUCT Structure

The TRANSLATE_STRUCT structure contains the translation table information. This structure is used for retrieving and setting the translation table information.

The following figure represents the TRANSLATE_STRUCT structure:

```
struct  Translate_Struct
{
  TCHAR  Filename[MAX_OBJECT_NAME+1];
  BYTE   Table[256];
  TCHAR  MsgId[MAX_MESSAGE_ID+1];
  int    ConditionCode;
  int    Feedback;
  TCHAR  MsgText[MAX_MESSAGE_TEXT+1];
  TCHAR  MsgData[MAX_MESSAGE_DATA+1];
};
typedef struct Translate_Struct  TRANSLATE_STRUCT;
```

Members

The TRANSLATE_STRUCT structure contains the following members:

Member	Description
FileName [MAX_OBJECT_NAME+1]	Specifies the name of the file where the translation information is stored.
Table [256]	Specifies the actual translation table information.

Member	Description
MsgId[MAX_MESSAGE_ID+1]	Specifies the message identifier that uniquely identifies a message.
ConditionCode	Specifies the return code that accompanies a message.
Feedback	Specifies additional return code information.
MsgText[MAX_MESSAGE_TEXT+1]	Specifies the message text.
MsgData[MAX_MESSAGE_DATA+1]	Specifies the message substitution field.

Appendix B

Return Codes

This appendix contains a list of return code values returned from the Connect:Direct C++ Class and the C applications programming interface functions. The following table lists the return codes defined in the CDAPI.H header file:

Name	Description
CD_NO_ERROR	No error detected.
CD_ENDOFDATA	No more data available.
CD_PARM_ERROR	Invalid parameter detected.
CD_INITIALIZE_ERROR	Initialization failed or initialization has not been performed.
CD_CONNECT_ERROR	Error occurred during attach processing.
CD_CONNECT_CANCELLED	Attach operation cancelled by the user.
CD_CONNECTED_ERROR	Invalid Connect:Direct server name.
CD_DISCONNECT_ERROR	Connect:Direct server disconnected from the client.
CD_NODENAME_ERROR	The Name field not set and the default not found.
CD_USERID_ERROR	Invalid user ID specified.
CD_ADDRESS_ERROR	Invalid TCP/IP address.
CD_PROTOCOL_ERROR	Invalid or unsupported protocol specified.
CD_HANDLE_ERROR	Invalid handle.
CD_HANDLE_TYPE_ERROR	The wrong handle type specified.
CD_LOGON_ERROR	Error while logging on to the Connect:Direct server. The user ID or password may be invalid.
CD_DIALOG_ERROR	Dialog box not created correctly.

Name	Description
CD_CANCEL	An error occurred creating the dialog box or retrieving the entered information.
CD_BUSY_ERROR	Operation failed. Connection is currently busy.
CD_IDLE_ERROR	Operation failed. Connection is currently idle.
CD_KQV_ERROR	Invalid KQV stream detected.
CD_NOT_FOUND	Object not found.
CD_ALREADY_EXISTS	Object already exists.
CD_ALLOCATE_ERROR	Allocation error occurred.
CD_NODE_ERROR	Invalid network map node.
CD_PARSER_ERROR	Parser detected an error.
CD_ACCESS_DENIED	Object access denied.
CD_SEND_ERROR	Error while sending error.
CD_RECEIVE_ERROR	Error while receiving error.
CD_CONNECTION_ERROR	A connection error occurred.
CD_REGISTRY_ERROR	An error occurred while opening the Registry.
CD_TIMEOUT_ERROR	Time-out value was reached.
CD_BUFFER_ERROR	The buffer is not big enough to hold all of the items in the list.
CD_COMMAND_ERROR	The command was not recognized.
CD_PROCESS_ERROR	The Process status is HE, held in error.
CD_UNDEFINED_ERROR	An unknown exception.
CD_NOT_SUPPORTED	An unknown exception.

Glossary

A

Adjacent Node

An entry in the network map that defines a Connect:Direct node with which the local Connect:Direct node can communicate. The adjacent node is also referred to as a remote node.

Advanced Program-To-Program Communication (APPC)

The general facility characterizing the LU 6.2 architecture and its various implementations in products.

Application Program Interface (API)

A Connect:Direct component that accepts commands and places them in an executable format.

C

Checkpoint Restart

Feature that eliminates the need to retransmit an entire file in the event of a transmission failure. If a copy procedure is interrupted, Connect:Direct restarts that copy at the last checkpoint.

Commands

Initiate and monitor activity within the Connect:Direct system.

Configuration Registry

A database of configuration information central to Windows operation. The Registry is the storage location for configuration settings for Windows applications. The Registry centralizes all Windows settings and provides security and control over system, security, and user account settings.

D

Domain Name Service (DNS)

A distributed database that provides a hierarchical naming system for identifying Internet hosts.

L

Local Node Record

The base record in a parameters file that defines the Connect:Direct server. It includes the most commonly used settings at a site and is the central node through which all communication is filtered. Depending upon how each remote node record is configured, trading partner node records can use settings that are defined in the local node record.

Logical Unit (LU) 6.2

A type of logical unit that supports general communication between programs in a distributed processing environment. LU 6.2 is characterized by (a) a peer relationship between session partners, (b) efficient utilization of a session for multiple transactions, (c) comprehensive end-to-end error processing, and (d) a generic application program interface (API) consisting of structured verbs that are mapped into a product implementation.

N

Network Map (Netmap)

The file that identifies all valid Connect:Direct nodes in a network including a local node record and a remote node record for each trading partner. The network map also defines the rules or protocols used by each node when communicating with the local Connect:Direct node.

Node

Any site in a network from which information distribution can be initiated.

New Technology File System (NTFS)

A system that uses fixed disks to organize data. NTFS treats files as objects with user and system defined attributes.

P

Primary Node (PNODE)

The node that submits the Connect:Direct Process to the secondary node (SNODE). In every communication, you must have a PNODE and an SNODE.

Process

A series of statements that initiate Connect:Direct activity, such as copying files and running jobs.

Process Statements

Instructions for transferring files, running operating system jobs, executing programs, or submitting other Connect:Direct Processes. They are used to build a Connect:Direct Process.

R

Registry

See Configuration Registry.

Remote Node

An entry in the network map that defines a Connect:Direct node with which the local Connect:Direct node can communicate. The remote node is also referred to as an adjacent node.

Retry Interval

The interval at which retries are performed as a part of the checkpoint-restart feature.

S

Secondary Node (SNODE)

The Connect:Direct node that interacts with the primary node (PNODE) during Connect:Direct Process execution and is the noncontrolling node. Every Process has one secondary node and one primary node.

Statistics File

Holds Connect:Direct statistics records that document the history of a Process.

Statistics Facility

Records Connect:Direct activities.

Systems Network Architecture (SNA)

The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks.

T

Transmission Control Queue (TCQ)

Holds information about Connect:Direct Processes that are currently executing or scheduled to execute in the future.

Index