

Developing Distributed Synergy Applications Using *xfNetLink* & *xfServerPlus*

Version 9.3



Printed: December 2009

The information contained in this document is subject to change without notice and should not be construed as a commitment by Synergex. Synergex assumes no responsibility for any errors that may appear in this document.

The software described in this document is the proprietary property of Synergex and is protected by copyright and trade secret. It is furnished only under license. This manual and the described software may be used only in accordance with the terms and conditions of said license. Use of the described software without proper licensing is illegal and subject to prosecution.

© Copyright 1998, 1999, 2001–2009 by Synergex

Synergex, Synergy, Synergy/DE, and all Synergy/DE product names are trademarks of Synergex.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

x/NetLink Java includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at <http://oss.software.ibm.com/icu4j>.

Windows, Visual Basic, and Visual Studio are registered trademarks of Microsoft Corporation.

All other product and company names mentioned in this document are trademarks of their respective holders.

DCN NL-01-9301

Synergex
2330 Gold Meadow Way
Gold River, CA 95670 USA

<http://www.synergex.com>

tel 916.635.7300

fax 916.635.6549

Introduction

Components Required for Distributed Computing with Synergy	xiv
Other Resources.....	xv
Product Support Information	xv
Synergex Professional Services Group	xvi
Comments and Suggestions.....	xvi

Part I: *xfServerPlus*

1. Preparing Your Synergy Server Code

Modularizing Your Code	1-1
Attributing Your Code	1-2
Using ELBs and Shared Images	1-3
Defining Logicals	1-4
Removing User Interface Elements	1-5
UI Toolkit Routines	1-5
Specifying a Base Channel Number.....	1-7
Passing Structures as Parameters.....	1-8
How Overlays Are Handled	1-9
Handling Variable-Length and Large Data.....	1-12
Passing a Single Parameter as a Memory Handle.....	1-12
Passing a System.String Parameter	1-13
Returning a Collection of Structures	1-14
Passing a System.Collections.ArrayList Parameter	1-16
Passing Arrays Larger Than 64K.....	1-17
Passing Binary Data	1-18
Handling Errors.....	1-19
Testing Your Synergy Code.....	1-20
Generating Test Skeletons with Workbench	1-21
Generating Test Skeletons from the Command Line	1-23
The gensyn Utility.....	1-24
Using the Test Skeletons	1-25
Using the <i>xfServerPlus</i> Application Program Interface	1-30
SET_XFPL_TIMEOUT.....	1-31
XFPL_LOG	1-33
XFPL_REGCLEANUP	1-34

2. Defining Your Synergy Methods

Understanding Routine Name, Method Name, and Method ID	2-2
Using Attributes to Define Synergy Methods	2-3
General Procedure	2-4
xfMethod Attribute	2-8
xfParameter Attribute	2-12
Attribute Examples	2-16
Documentation Comments.....	2-19
Using the MDU to Define Synergy Methods	2-22
Creating New Methods.....	2-22
Specifying a Method ID	2-28
Defining Parameters	2-28
Modifying Methods and Parameters	2-35
Deleting Data from the SMC	2-36
Searching for Methods and Parameters	2-37
Setting the Catalog Location.....	2-38
Importing and Exporting Methods	2-39
Verifying Repository Structure Sizes and Enumerations	2-42
Defining Multiple Synergy Method Catalogs.....	2-43
Creating New SMC Files	2-43
Specifying Which SMC to Update.....	2-45
Setting the XFPL_SMCPATH Environment Variable for xfServerPlus	2-46
The Method Definition Utility.....	2-50
The SMC/ELB Comparison Utility.....	2-55
Windows and UNIX	2-55
OpenVMS	2-57
Running an SMC/ELB Comparison from Workbench	2-58

3. Configuring and Running xfServerPlus

The Big Picture	3-1
Running xfServerPlus	3-3
Running xfServerPlus on Windows	3-3
Running xfServerPlus on UNIX	3-9
Running xfServerPlus on OpenVMS	3-12
Testing xfServerPlus	3-15
xfServerPlus Status Codes	3-16
Using the xfpl.ini File	3-18
Using an Alternate xfpl.ini File.....	3-18
Setting the XFPL_INIPATH Environment Variable	3-19
Configuring Compression	3-22

Using Encryption	3-23
Setting up the <i>xfServerPlus</i> Machine for Encryption.....	3-24
Setting up the <i>xfNetLink</i> Machine for Encryption.....	3-26
Specifying the Data to Encrypt for Slave Encryption.....	3-28
Using Server-Side Logging	3-29
Setting Options for the <i>xfServerPlus</i> Log	3-30
Error Messages in the <i>xfServerPlus</i> Log	3-39
Debugging Your Remote Synergy Routines	3-40
Debugging Remote Synergy Routines via Telnet	3-42
Deploying Your Distributed Application	3-44
Deploying the Server	3-44
Deploying the Client	3-45
Configuring <i>xfServerPlus</i> for Remote Data Access	3-45
Remote Data Access When <i>xfServerPlus</i> Is on Windows.....	3-46
Remote Data Access When <i>xfServerPlus</i> Is on UNIX.....	3-48

Part II: *xfNetLink* Synergy Edition

4. *xfNetLink* Synergy: Overview

System Overview	4-1

5. Configuring & Testing *xfNetLink* Synergy

The Big Picture.....	5-1
Configuring <i>xfNetLink</i> Synergy	5-2
Specifying the Host Name and Port Number.....	5-3
Specifying Time-out Values.....	5-3
Specifying Debug Options	5-6
Testing <i>xfNetLink</i> Synergy	5-6

6. Calling Synergy Routines Remotely from Synergy

Making Remote Calls	6-1
Making Remote Calls with %RXSUBR	6-1
Making Remote Calls Using a Routine Call Block	6-3
Handling Errors.....	6-3
Troubleshooting Techniques	6-5
Running an <i>xfServerPlus</i> Session in Debug Mode	6-5
Viewing Packets.....	6-9

7. *xfNetLink* Synergy API

%RX_CONTINUE	7-2
%RX_DEBUG_INIT	7-4
%RX_DEBUG_START	7-5

RX_GET_ERRINFO	7-7
RX_GET_HALTINFO	7-9
%RX_RMT_ENDIAN	7-11
%RX_RMT_INTSIZE	7-12
%RX_RMT_OS	7-13
%RX_RMT_SYSINFO	7-14
%RX_RMT_TIMEOUT	7-15
RX_SETRMTFNC	7-16
RX_SHUTDOWN_REMOTE	7-17
%RX_START_REMOTE	7-18
%RXSUBR	7-23

Part III: *xfNetLink* Java Edition

8. *xfNetLink* Java: Overview

System Requirements	8-1
Technical Overview of <i>xfNetLink</i> Java and <i>xfServerPlus</i>	8-1
The Client	8-2
The Web Server	8-2
The Synergy Host	8-3
Creating a Connection	8-4
Calling a Synergy Method	8-4
The Synergy Method Catalog	8-5

9. Creating Java Class Wrappers

The Big Picture	9-1
Setting the Classpath	9-3
Creating a Java JAR File in Workbench	9-4
Creating a Synergy/DE Java Component Project	9-4
Generating Java Class Wrappers	9-7
Building the JAR File	9-8
Creating a Java JAR File from the Command Line	9-8
The genxml Utility	9-9
The genjava Utility	9-12
Building the JAR File	9-13
Understanding the Generated Classes	9-14
Procedural Classes	9-14
Structure Classes	9-15
Editing the Java Source Files	9-16
Generating Javadoc	9-16

10. Calling Synergy Routines from Java

Deploying Your Distributed Application	10-1
Deploying the Server	10-1
Deploying the Client	10-1
Configuring the <i>xfNetLink</i> Java Properties File	10-2
Creating and Naming a Properties File	10-4
Using a Properties File vs. Using the “set” Methods	10-5
Specifying the Host Name and Port Number	10-6
Specifying a Port Number Range	10-7
Specifying Logging Options	10-8
Specifying Time-out Values	10-8
Using Your JAR File	10-11
Using Structures	10-15
Passing Binary Data	10-17
Setting a Call Time-Out	10-19
Writing to the <i>xfServerPlus</i> Log	10-20
Understanding Java Connection Pooling	10-20
Implementing Pooling	10-22
Using Your JAR File with Connection Pooling	10-22
Setting Up a Pooling Properties File	10-24
Pool Maintenance	10-31
Using the Pooling Support Methods	10-32
Method Reference	10-35

11. Error Handling and Troubleshooting in *xfNetLink* Java

Handling Exceptions	11-1
Exceptions in <i>xfNetLink</i> Java	11-2
Troubleshooting Techniques	11-7
Client-Side Logging	11-7
Testing <i>xfNetLink</i> Java	11-9
Running an <i>xfServerPlus</i> Session in Debug Mode	11-11

12. *xfNetLink* Java Class Reference

<i>Synergex.net.SynProxyNetException</i>	12-3
<i>Synergex.net.xfServerNackException</i>	12-4
<i>Synergex.net.xfServerPlusUnavailableException</i>	12-6
<i>Synergex.synProxy.BadSynPacketException</i>	12-7
<i>Synergex.synProxy.SynergyWebProxy</i>	12-8
<i>Synergex.synProxy.SynProxyBadPropsException</i>	12-17
<i>Synergex.synProxy.SynProxyException</i>	12-18
<i>Synergex.synProxy.SynProxyIncompatibleServerException</i>	12-19
<i>Synergex.synProxy.xfLaunchFatalException</i>	12-20

Contents

- Synergex.synProxy.xfRemoteExecException 12-22
- Synergex.synTypes.ArrayDescriptor 12-24
- Synergex.synTypes.FieldDescriptor 12-25
- Synergex.synTypes.ImpDecDescriptor 12-26
- Synergex.synTypes.IStructManager 12-27
- Synergex.synTypes.ScalarDescriptor 12-28
- Synergex.synTypes.StructDescriptor 12-29
- Synergex.synTypes.synAlpha 12-30
- Synergex.synTypes.synArray 12-32
- Synergex.synTypes.synArrayList 12-36
- Synergex.synTypes.synByteArray 12-37
- Synergex.synTypes.synDec 12-39
- Synergex.synTypes.SynergyType 12-42
- Synergex.synTypes.synHandle 12-44
- Synergex.synTypes.synImpDec 12-46
- Synergex.synTypes.synNull 12-48
- Synergex.synTypes.SynProxyBadTypeException 12-49
- Synergex.synTypes.SynProxyInvalidStructException 12-50
- Synergex.synTypes.SynProxyMixedTypeArrayException 12-51
- Synergex.synTypes.synStruct 12-52
- Synergex.util.SWPCConnect 12-53
- Synergex.util.SWPManager 12-55
- Synergex.util.xfJCWException 12-57
- Synergex.util.xfPoolException 12-58

Part IV: xfNetLink COM Edition

13.Creating Synergy COM Components

- System Overview 13-1
- The Big Picture 13-3
- Creating and Registering a Type Library in Workbench..... 13-5
 - Creating a Synergy/DE COM Component Project..... 13-5
 - Generating a Type Library 13-8
 - Registering a Type Library 13-8
 - Unregistering a Type Library 13-9
 - Viewing a Type Library in Workbench..... 13-9
- Creating and Registering a Type Library from the Command Line 13-11
 - The genxml Utility 13-11
 - The gentlb Utility 13-15
 - The regtlb Utility 13-16

14. Calling Synergy Routines from COM

Deploying Your Distributed Application	14-1
Deploying the Server	14-1
Deploying the Client	14-1
Using the Synergy Type Library Configuration Utility	14-2
Using Your Type Library	14-6
Using Structures	14-10
Setting a Call Time-Out	14-12
Writing to the <i>xfServerPlus</i> Log	14-12
Understanding COM Pooling	14-14
Connection Pooling	14-14
Object Pooling	14-14
Reusing Connections and Objects	14-15
Implementing Pooling	14-16
Setting Up the Client Machine	14-17
Changing the Pool Configuration	14-21
Using the Pooling Support Methods	14-22
Programming Tips for Object Pooling	14-25
Method Reference	14-27

15. Error Handling and Troubleshooting in *xfNetLink* COM

Handling Errors	15-1
Troubleshooting Techniques	15-9
Using Client-Side Logging	15-9
Testing <i>xfNetLink</i> COM	15-11
Running an <i>xfServerPlus</i> Session in Debug Mode	15-12

Part V: *xfNetLink* .NET Edition

16. Creating Synergy .NET Assemblies

System Requirements	16-1
System Overview	16-2
The Big Picture	16-3
Creating an Assembly in Workbench	16-5
Creating a Synergy/DE .NET Component Project	16-6
Controlling the .NET Environment	16-10
Generating C# Classes	16-10
Building the Assembly	16-11
Creating an Assembly from the Command Line	16-13
The <i>genxml</i> Utility	16-13
The <i>gens</i> Utility	16-16
Building the Assembly	16-20

Understanding the Generated Classes.....	16-22
Procedural Classes	16-22
Structure Classes.....	16-23
DataTable Classes	16-23
Enumerations	16-24
Custom Attributes	16-24
Using Your Own Key File.....	16-25
Editing the Generated Files.....	16-26
Editing Information in AssemblyInfo.cs	16-26
Generating API Documentation.....	16-27
Adding Documentation Comments	16-27
Generating an XML File	16-28
Creating the API Documentation.....	16-29

17. Calling Synergy Routines from .NET

Deploying Your Distributed Application	17-1
Deploying the Server	17-1
Deploying the Client (for Development)	17-1
Deploying the Client (at a Customer Site)	17-2
Using an Application Configuration File	17-3
Creating and Editing Configuration Files	17-5
Using Your Synergy .NET Assembly.....	17-9
Using Multiple Copies of the Same Class	17-12
Using Structures	17-14
Using DataTables.....	17-18
Using Enumerations	17-21
Passing Binary Data	17-22
Setting a Call Time-Out	17-24
Writing to the xfServerPlus Log	17-24
Understanding .NET Pooling	17-26
Object Pooling.....	17-26
Reusing Objects	17-27
Tips for Creating Poolable Objects.....	17-27
Implementing Pooling	17-28
Implementation Overview	17-28
Setting Up the Client Machine	17-28
Changing the Pool Configuration	17-31
Using the Pooling Support Methods.....	17-32
Writing Code That Uses Pooled Objects	17-35
Method Reference	17-38
Procedural Methods.....	17-38
Structure Methods	17-40
DataTable Methods.....	17-40

18. Error Handling and Troubleshooting in *xfNetLink* .NET

Handling Errors	18-1
Troubleshooting Techniques	18-7
Using Client-Side Logging	18-7
Testing <i>xfNetLink</i> .NET	18-9
Running an <i>xfServerPlus</i> Session in Debug Mode	18-10

Appendices**A. Configuration Settings**

<i>xfServerPlus</i>	A-1
<i>xfNetLink</i> Synergy	A-3
<i>xfNetLink</i> Java	A-4
<i>xfNetLink</i> COM	A-7
<i>xfNetLink</i> .NET	A-8

B. Data Type Mapping

<i>xfNetLink</i> Java	B-1
<i>xfNetLink</i> COM	B-8
<i>xfNetLink</i> .NET	B-12

C. *xfNetLink* Synergy Sample Code

Client Application (syncIt.dbl)	C-1
Server-Side Code (HELLO Subroutine)	C-4
Running the Hello Program	C-4

D. *xfNetLink* Java Sample Code

Client Application (hello.jsp and hello.java)	D-1
Server-Side Code (HELLO Subroutine)	D-4
Running the Hello Program	D-4

Glossary**Index**

x/NetLink and *x/ServerPlus*, part of the *x/Series*[™] family, enable you to access Synergy routines and data remotely from a Synergy[™], Java[™], COM, or .NET client. *x/NetLink* provides four methods for accessing Synergy logic with an assortment of clients.

- *x/NetLink* Synergy Edition is a set of routines distributed with Synergy/DE Professional Series. These routines work in conjunction with *x/ServerPlus* to execute Synergy routines stored on a remote machine. The user interface is in a Synergy application on the client machine. Using *x/NetLink* Synergy, you can create full-scale multi-tier applications or incrementally improve the performance of existing applications.
- *x/NetLink* Java Edition works in conjunction with Sun Microsystems' Java programming language or JavaScript[™] scripting language. Using the component generation tools, you can generate Java class wrappers and build them into a JAR file which will reference your Synergy routines. The user interface can be presented by a Java application, a Java applet, or JavaServer Pages[™].
- *x/NetLink* COM Edition enables a client, using a standard COM calling mechanism, to call Synergy routines residing on a remote server machine. Using the component generation tools, you can create a type library that references Synergy routines. The user interface can be written in any language that supports COM, such as Visual Basic® or Active Server Pages. *x/NetLink* COM is supported on Windows platforms only; however, you can create a COM component from Synergy routines that reside on UNIX and OpenVMS.
- *x/NetLink* .NET Edition enables a .NET client to call Synergy routines residing on a remote server machine. Using the component generation tools, Microsoft® Visual Studio®, and the .NET Framework SDK, you can create an assembly that references Synergy routines. *x/NetLink* .NET is supported on Windows platforms only; however, you can create an assembly from Synergy routines that reside on UNIX and OpenVMS.

Components Required for Distributed Computing with Synergy

To build a distributed computing system with Synergy/DE™, you'll need the items listed below.

- ▶ Professional Series Development Environment or Professional Series Workbench
- ▶ *xfNetLink*. You'll need one of the following:
 - ▶ *xfNetLink Synergy*: Synergy routines for connecting a Synergy client to a remote Synergy server.
 - ▶ *xfNetLink Java*: tools to generate Java class wrappers and create a JAR file; Java classes for connecting a Java client to a remote Synergy server.
 - ▶ *xfNetLink COM*: tools to generate Synergy type libraries for connecting a COM-compliant client to a remote Synergy server.
 - ▶ *xfNetLink .NET*: tools to generate .NET assemblies for connecting a .NET client application to a remote Synergy server.
- ▶ *xfServerPlus*, which includes the following:
 - ▶ *xfServerPlus* for remote execution of Synergy routines. You must have at least one *xfServerPlus* license for each machine you want to use as an application logic server.
 - ▶ The Method Definition Utility (MDU) for defining Synergy functions and subroutines that you wish to call remotely in the Synergy Method Catalog.
 - ▶ *xfServerPlus* Application Program Interface (API) for setting remote execution options.



If you are using *xfNetLink Java*, *xfNetLink COM*, or *xfNetLink .NET* you'll need additional software, such as the Java Development Kit, a web server, or the .NET Framework SDK.

Other Resources

These resources may be useful as you use *xfNetLink* and *xfServerPlus*:

- ▶ *Modularizing Your Synergy Code: The First Step to Distributed Computing*. This white paper is available on the Online Manuals CD.
- ▶ Tutorials for the *xfNetLink* products. Check the Synergex™ web site to see which tutorials are currently available.
- ▶ Release notes for *xfNetLink* Java, *xfNetLink* COM, *xfNetLink* .NET, and *xfServerPlus*.
- ▶ Synergy Language release notes for information about *xfNetLink* Synergy.
- ▶ *Installation Configuration Guide* for information about the **rsynd** program.
- ▶ *Getting Started with Synergy/DE* for information about using Professional Series Workbench.

Product Support Information

If you cannot find the information you need in this manual or in the resources listed above, you can call Synergy/DE Developer Support at 800.366.3472 (in North America) or 916.635.7300 for assistance. To purchase Synergy/DE Developer Support services, call your Synergy/DE account manager at the above phone numbers.

Before you contact us, make sure you have the following information:

- ▶ Version of Synergy/DE, *xfServerPlus*, and *xfNetLink* you are running.
- ▶ Name and version of the operating systems and hardware platforms you are running for *xfServerPlus*, the *xfNetLink* client, and (if applicable) the web server.
- ▶ Error number and associated error text if you need help with a Synergy/DE, *xfNetLink*, or *xfServerPlus* error.
- ▶ Statement at which the error occurred.
- ▶ Exceptions thrown by *xfNetLink* Java or *xfNetLink* .NET, or errors thrown by *xfNetLink* COM.
- ▶ Contents of the *xfServerPlus* log or the application event log (Windows) or syslog (UNIX).
- ▶ Contents of the client-side log.

Introduction

Synergex Professional Services Group

- ▶ Exact steps that preceded the problem.
- ▶ Anything that changed (e.g., code, data, hardware) before this problem occurred.
- ▶ Whether the problem happens every time, and whether it is reproducible in a small test program.

Synergex Professional Services Group

If you would like assistance implementing new technology or would like to bring in additional experienced resources to complete a project or customize a solution, Synergex Professional Services Group (PSG) can help. PSG provides comprehensive technical training and consulting services to help you take advantage of Synergex's current and emerging technologies. For information and pricing, contact your Synergy/DE account manager at 800.366.3472 (in North America) or 916.635.7300.

Comments and Suggestions

We welcome your comments and suggestions for improving this manual. Send your comments, suggestions, and queries, as well as any errors or omissions you've discovered, to doc@synergex.com.

Part I: *xf*ServerPlus ▶

Preparing Your Synergy Server Code

`xfNetLink` and `xfServerPlus` enable you to access existing Synergy Language code within your Synergy applications from a remote client. You can call both functions and subroutines remotely. However, your Synergy server code may require some alterations in order to work effectively with `xfNetLink` and `xfServerPlus`: it should be modularized, routines must be contained in executable libraries (ELBs) or shared images (on OpenVMS), and you will need to remove all user interface elements, including error messages.

This chapter gives an overview of changes you may need to make to your Synergy server code and includes information on passing different types of data and testing your modularized Synergy code. It also describes the Synergy Language subroutines that you can use to set remote execution options for `xfServerPlus`.

Modularizing Your Code

Modular code is contained in an isolated functional unit with a well-defined, published interface (i.e., an argument list). The published interface tells the programmer how to use the code, and all functionality is provided through that interface.

In modular code, routines are usually short and perform single tasks. They do not rely on global data to accomplish their tasks. Data is passed via arguments, and global or common data is kept to a minimum.

For more information on designing a modular distributed application, see “[Design Considerations](#)” in the “Accessing Logic Remotely with `xfServerPlus`” chapter of *Getting Started with Synergy/DE*. For detailed information on modular code and suggestions for modularizing your Synergy Language code, see the Synergex white paper, *Modularizing Your Synergy Code: The First Step to Distributed Computing*, available on the Online Manuals CD.

Attributing Your Code

Attributes (introduced in version 9.3) are a feature of Synergy Language that enable you to automate the population and maintenance of the Synergy Method Catalog (SMC). The SMC identifies the Synergy routines that you have prepared for remote calling. The most efficient way to populate the SMC with details about your routines is by attributing your code. (You can also populate the SMC by entering data manually using the Method Definition Utility.)

xfServerPlus supports two attributes—*xfMethod* and *xfParameter*—each of which has a number of properties that are used to describe your Synergy routines. The *xfMethod* attribute is required, but *xfParameter* may be optional, depending on the type of parameter you are defining. The example below shows a simple *xfMethod* attribute statement for the function *ReturnError*.

```
{xfMethod(interface="ConsultApp", elb="EXE:Consult")}  
function ReturnError ,string  
    req in userToken          ,a22  
endparams
```

Once you have added attributes to your Synergy code, you will run the **dbl2xml** utility using one or more Synergy source files as input. (Some additional changes to your code may be necessary before you can use **dbl2xml**. For example, you may need to add parameter modifiers [IN, OUT, REQ, etc.], as shown in the example above.) The **dbl2xml** utility outputs an XML file containing information about your routines. This XML file is then imported into the SMC.

All of the options for defining routines and parameters that are available in the MDU, including documentation comments, are also available when you attribute your code. Attributes can be used regardless of the *xfNetLink* client you are using.

For more information about the SMC, see [chapter 2, “Defining Your Synergy Methods”](#). For detailed instructions on attributing your code, see [“Using Attributes to Define Synergy Methods” on page 2-3](#). For the **dbl2xml** syntax, see [“The dbl2xml Utility”](#) in the “General Utilities” chapter of *Synergy Language Tools*.

Using ELBs and Shared Images

Your Synergy server routines must reside in one or more ELBs or shared images. (Although we use the term “ELB”, the information in this section applies to both ELBs and shared images, except where noted.) When you define your routines in the Method Definition Utility or attribute your code, you will include the name of the ELB, so that *xjServerPlus* can find the routine. The ELBs are opened and closed by *xjServerPlus*.

Considerations and restrictions

- ▶ If you have a Synergy routine in one ELB that calls a routine in a second (dependent) ELB, the dependent ELB will not be opened by *xjServerPlus*. (Note that the dependent ELB may be your own ELB or an ELB distributed with Synergy/DE, such as **tklib.elb**.) To ensure that the dependent ELB is opened, do one of the following:
 - ▶ (Windows and UNIX only) When you create the primary ELB, link it against the dependent ELB(s). The dependent ELBs will be opened when the primary ELB is opened. For details on linking ELBs, see [“Invoking the linker on Windows and UNIX”](#) in the “Building and Running Synergy Language Programs” chapter in *Synergy Language Tools*.
 - ▶ (OpenVMS) When you create the primary shared image, link it against the secondary shared image. Note that if the call to the routine in the secondary shared image is made directly or with XCALL, you’ll get an error if the secondary shared image is not linked. However, if the call is made with XSUBR, you won’t get an error, and will simply have to remember to link the shared images or use one of the other options listed below to access the routine in the secondary shared image.
 - ▶ Modify the routine in the primary ELB to open the dependent ELB before calling any routines in the dependent ELB.
 - ▶ Call an initialization routine that opens all the ELBs that will be needed.
 - ▶ Put all routines in a single ELB.
- ▶ You can include an initialization routine that sets up global data, as long as the global data is contained within the ELB.
- ▶ Do not chain from within an ELB.
- ▶ Do not use the EXEC subroutine to call a non-Synergy Language program from within an ELB.

Preparing Your Synergy Server Code

Using ELBs and Shared Images

- ▶ Routines in an OpenVMS shared image must not contain any XCALLs to routines that require the command line interpreter. These include PURGE, RUNJB (unless it creates another detached process, in which case it may be used), SETLOG, SHELL, SPAWN, VMCMD, and any OpenVMS library routine that requires a CLI be present.

For more information on ELBs and shared images, see the [“Building and Running Synergy Language Programs”](#) chapter in *Synergy Language Tools*.

Defining Logicals

If you use logicals to point to the directories that your ELBs or shared images reside in, and you use those logicals when defining routines in the Synergy Method Catalog, you must define the logicals so that *xjServerPlus* knows how to resolve them.

Defining logicals on Windows and UNIX

On Windows and UNIX, define the logicals in the **xfpl.ini** file, which is located by default in DBLDIR. (Note that it is possible to specify a different location for **xfpl.ini**, as well as to have more than one **xfpl.ini** file; see [“Using the xfpl.ini File”](#) on page 3-18.)

The maximum length for an XFPL_LOGICAL translation value is 1,024 characters.



If any of your ELBs are linked to dependent ELBs (see [page 1-3](#)), you must define the logicals for both the primary and the dependent ELBs in the **xfpl.ini** file.

Define logicals like this:

```
XFPL_LOGICAL: LOGICAL_NAME=path
```

Defining logicals on OpenVMS

On OpenVMS, we recommend defining logicals in the file `DBLDIR:SERVER_INIT.COM`.

- ▶ To specify logicals for system-wide visibility:
`$ DEFINE/SYS LOGICAL_NAME value`
- ▶ To specify logicals for a specific instance of *xfServerPlus*:
`$ DEFINE/TABLE=LNMRSDMSMGR_port /USER LOGICAL_NAME-value`

where *port* is the port number on which the specific *xfServerPlus* session is listening.

For additional information about defining logicals on OpenVMS, see “[Defining logical names](#)” in the “Configuring *xfServer*” chapter of the *Installation Configuration Guide*.

Removing User Interface Elements

xfServerPlus runs as a background process using the **db**s service runtime, which does not include support for console operations. Consequently, Synergy routines that you prepare for remote access should not include user interface elements. (The user interface will be located on the client machine.) If any of the Synergy routines you want to access remotely require input from or send messages to the user, use the Synergy windowing API, or may generate untrapped errors, they should be adjusted to work as server-level logic. (See “[The service runtimes](#)” in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools* for more information about **db**s, including specifics on functionality that is limited or unavailable.)

UI Toolkit Routines

In general, you should remove UI Toolkit routines from code that you plan to call remotely. Do not use routines that require use of the computer console (e.g., `U_MESSAGE`, `U_POPUP`) or that create or use windows (e.g., `I_LDINP`, `T_PUTTXT`). However, you can use `U_START` and `U_FINISH`, as well as file I/O and channel maintenance routines such as `U_OPEN` and `U_CLOSE`.

Preparing Your Synergy Server Code

Removing User Interface Elements

To use Toolkit routines with *xfServerPlus*, you must set the WND and SYNTXT environment variables in the **xfpl.ini** file (**DBLDIR:SERVER_INIT.COM** on OpenVMS), so that *xfServerPlus* knows how to resolve them. See [“Defining Logicals” on page 1-4](#) for instructions. In addition, you must explicitly open the Toolkit library, **tklib.elb**, with a call to **OPENELB** or link **tklib.elb** to your ELB (Windows and UNIX only). See **OPENELB** in the “System-Supplied Subroutines, Functions, and Classes” chapter of the *Synergy Language Reference Manual* for details.

If you use the **U_START** subroutine, you need to **.INCLUDE** the **tools.def** file. To do this, include the following code in the routine that calls **U_START**:

```
.define TOOLS_INIT
.include "WND:tools.def"
```

You may want to exclude the global data sections of the **tools.def** file by defining **D_NO_GLOBAL_DATA** prior to including **tools.def**. This means that only the definitions and function declarations from **tools.def** will be included. For example:

```
.define D_NO_GLOBAL_DATA
.include "WND:tools.def"
```

U_START closes all channels in use from 1 through 255 if the *first_channel* and *last_channel* arguments are not set. If those arguments are set, it closes only channels in use in the range defined by those arguments. Either way, there’s a chance that **U_START** will close the channels in use by the Synergy Method Catalog files or the **XFPL_LOG** routine, which may result in an “Invalid operation for file type” error (**\$ERR_FILOPT**). There are two possible solutions to this problem:

- ▶ If *xfServerPlus* is using the default base channel (243; see next page) you can set *last_channel* to a value less than 243.
- ▶ If you are using **XFPL_BASECHAN** to specify a non-default base channel number for *xfServerPlus*, set the value for **XFPL_BASECHAN** outside the range of channels specified by the *first_channel* and *last_channel* arguments. (See [“Specifying a Base Channel Number”](#) below for more information.)

Specifying a Base Channel Number

By default, *xyServerPlus* uses 243 as the base channel number. This is the channel number that *xyServerPlus* tries first whenever it needs to open a channel.

For example, when opening the SMC files (**cdt.ism** and **cmpdt.ism**), *xyServerPlus* attempts to use channels 243 and 244. If 243 is not available, it tries 244, then 245, and so on until an open channel is found.

If your Synergy server applications use hard-coded channels that are in the range used by *xyServerPlus*, you may receive a “Channel is in use” error (`$ERR_CHNUSE`). To avoid this problem, you can set `XFPL_BASECHAN` in the **xfpl.ini** file to specify that a different base channel number be used. Valid values are 2 through 254.

For example, if you set

```
XFPL_BASECHAN = 150
```

xyServerPlus will attempt to open the SMC files on channels 150 and 151. If 150 is currently in use, *xyServerPlus* will try 151, then 152, and so on until it finds an open channel. Should *xyServerPlus* reach channel 255 without finding an open channel, it will go to `XFPL_BASECHAN - 1` (149 in our example) and continue down from there to channel 1.

If you use hard-coded channels in your code, you should select a base channel number that is not likely to cause interference with your applications. You may also want to consider modifying your code to use available channels rather than hard-coded channels.

Passing Structures as Parameters

You can pass repository structures as parameters when using *x/NetLink* Java (with class wrappers) or when using *x/NetLink* COM or *x/NetLink* .NET. *x/NetLink* supports passing structures of primitive, date, and time types; structures with embedded structures (that is, groups or struct data types) or embedded arrays; and arrays of structures. User-defined fields are treated as strings, with one exception when using a .NET client. See [Appendix B](#) for complete information on supported data types and for details on how repository field data types are mapped from Synergy to *x/NetLink*.

Regardless of the type of Synergy component you create, during the component generation process, the repository structure becomes a class (named with the structure name), and each field in the structure becomes a property of that class. (In *x/NetLink* .NET, you can choose to generate repository structure fields as properties or as public fields. See [“Creating a Synergy/DE .NET Component Project” on page 16-6](#) or [“The gens Utility” on page 16-16](#).)

By default, the repository field name becomes the property name. To specify an alternate name, enter it in the Alternate name field in Repository. Then, indicate you want to use the alternate name by selecting the “Use alternate field names” option in the Component Information dialog box in Workbench, or by specifying the **-n** option when you run **genxml** from the command line.

When using structures with *x/NetLink*, the sum of the sizes of the fields in the structure must always equal the total size of the structure as calculated by Repository (this value displays at the bottom of the Field Definitions list in Repository). Similarly, the sum of the sizes of the fields in a group must equal the size of the group (if the group size is declared). In most circumstances, this will not be an issue. The only time it is likely to become a problem is when you are using the Excluded by Web flag (see [page 1-10](#)).



If you intend to attribute your Synergy code and use the XML file output by **dbl2xml** to populate the SMC, structures passed as ordinary parameters and as arrays must be defined as structfields in your Synergy code. See [“Structure”](#) in the “Defining Data” chapter of the *Synergy Language Reference Manual* for more information on structfields. Structures passed as ArrayLists and structure collections are identified by attributes. See [“Using Attributes to Define Synergy Methods” on page 2-3](#).

For more information on using structures with the individual *x/NetLinks*, refer to the following:

- ▶ Java, see [page 10-15](#)
- ▶ COM, see [page 14-10](#)
- ▶ .NET, see [page 17-14](#)

If your Java, COM, or .NET client passes structures as parameters, and you want to also use an *x/NetLink* Synergy client, you can do so without altering your server-side code. Simply create a separate SMC entry for the Synergy server routine and define the parameter as an alpha rather than a structure.

How Overlays Are Handled

If your structure definition in Repository includes overlays, you can either accept the default behavior or explicitly indicate which fields should be included/excluded by using the “Excluded by Web” flag when defining fields. The explanations below apply to both structures and groups.

The default behavior

By default, the way your fields are defined in Repository determines which fields will be accessible to someone using your Synergy component. The structure is read from the top down, and the “real” fields (overlaid fields)—rather than the overlays—are used. To determine whether you can use your repository structures as they are currently defined, you need to first determine which fields you want to make available in your client application.

For example, if you have this structure in your repository

```
customer
  firstname      ,a20
  mi             ,a1
  lastname       ,a30
  name           ,a51 @ firstname
```

the default behavior enables you to access the `firstname`, `mi`, and `lastname` fields from your client application. For example:

```
Customer.Firstname
Customer.Mi
Customer.Lastname
```

Preparing Your Synergy Server Code

Passing Structures as Parameters

However, if you have this structure in your repository

```
customer
  name          ,a51
  firstname     ,a20 @ name
  mi            ,a1  @ name + 20
  lastname      ,a30 @ name + 21
```

the default behavior enables you to access only the name field from your client application. For example:

```
Customer.Name
```

Using the “Excluded by Web” flag

If the default behavior is undesirable, you can use the “Excluded by Web” flag in Repository to explicitly indicate which fields are included/excluded. Fields for which the Excluded by Web flag is set will not be included in your Synergy component. Once you select the Excluded by Web flag for one or more fields in a structure, **genxml** will cease to use the default behavior for that structure and will honor the Excluded by Web flag instead.



The Excluded by Web flag is intended for use *only* with overlays and overlaid fields. Do not attempt to use it with other repository fields.

The sum of the sizes of the selected fields must always equal the size of the entire structure as calculated by Repository. Note that Repository *does not* recalculate the structure size when you exclude fields with the Excluded by Web flag. (The structure size, as calculated by Repository, is displayed at the bottom of the Field Definitions list. See “[Defining Fields](#)” in the “Repository” chapter of the *Repository User’s Guide*.)

For example, if you have this structure in your repository

```
customer
  name          ,a51
  firstname     ,a20 @ name
  mi            ,a1  @ name + 20
  lastname      ,a30 @ name + 21
```

and you want to be able to access the `firstname`, `mi`, and `lastname` fields like this from your client application

```
Customer.Firstname  
Customer.Mi  
Customer.Lastname
```

you would set the Excluded by Web flag for the name field.

► To prepare structures for passing as parameters

1. Define the structure in the repository or use an existing structure. See the “[Repository](#)” chapter of the *Repository User’s Guide* for information.
2. If your structures include overlays, decide how you want to handle them and set the “Excluded by Web” flag if necessary.
3. If you are using the MDU to populate the SMC,
 - Specify the repository when you start the Method Definition Utility (MDU). See “[Using the MDU to Define Synergy Methods](#)” on page 2-22.
 - Select the structures when you define parameters in the MDU. See [step 1 on page 2-29](#).
4. If you are attributing your code and using the XML file output by `dbl2xml` to populate the SMC, `.INCLUDE` the repository structure in your Synergy source code and define the structure as a structfield. See “[Using Attributes to Define Synergy Methods](#)” on page 2-3.



If you include structures in your parameter definitions in the SMC, and then later alter those structures in Repository, they are *not* automatically updated in the SMC. You can use the Verify Catalog utility in the MDU to update the structure sizes. See “[Verifying Repository Structure Sizes and Enumerations](#)” on page 2-42 for more information.

5. When you create your Synergy component, specify the repository in the Component Information dialog box in Workbench or on the command line when running `genxml`. For details, refer to the chapter for the *x/NetLink* client you are using:
 - Java, see [chapter 9, “Creating Java Class Wrappers”](#)
 - COM, see [chapter 13, “Creating Synergy COM Components”](#)
 - .NET, see [chapter 16, “Creating Synergy .NET Assemblies”](#)

Handling Variable-Length and Large Data

Normally, you must define the size of each parameter in your Synergy server routines in the Synergy Method Catalog (SMC) so that *xfServerPlus* knows what to expect. (See [chapter 2](#) for details on defining data in the SMC.) Sometimes, though, you may not know exactly what size the data will be, or you may know that it will vary in size, or your application may need to handle data that exceeds the usual size limits. If your application needs to support such cases, you may be able to use one of the following methods:

- ▶ Use a memory handle to pass a single, non-array parameter of variable length and/or larger than 64K. See [“Passing a Single Parameter as a Memory Handle”](#), below.
- ▶ (.NET client only) Use a Synergy `System.String` class to pass a single parameter of variable length and/or larger than 64K. See [“Passing a System.String Parameter”](#) on page 1-13.
- ▶ (Java and .NET clients only) Use a memory handle to pass a collection of structures, which can vary in the number of elements it contains, to an `ArrayList` on the client. See [“Returning a Collection of Structures”](#) on page 1-14.
- ▶ (.NET client only) Use a Synergy `System.Collections.ArrayList` class to pass an `ArrayList` of structures or other types of elements to the client or receive an `ArrayList` from the client. See [“Passing a System.Collections.ArrayList Parameter”](#) on page 1-16.
- ▶ Use a memory handle on a Synergy client to pass an array larger than 64K. (All of the *xfNetLink* clients can handle arrays larger than 64K, but the Java, COM, and .NET clients require no special technique to do so.) See [“Passing Arrays Larger Than 64K”](#) on page 1-17.

Passing a Single Parameter as a Memory Handle

Use this method to pass a non-array parameter that is of variable length and/or larger than 64K (65,535 bytes) in size. This method is supported on all *xfNetLink* clients.

Note the following:

- ▶ Your Synergy server routine must declare the argument that receives the data as a memory handle (`i4`; do not use `int`). *xfServerPlus* will place the data in a memory area and pass the memory handle allocated to that area to your

Synergy server routine. (You *must* use the memory handle provided by *xfServerPlus*; do not attempt to allocate your own.) After the data has been returned to *xfNetLink*, *xfServerPlus* will free the memory area.

- ▶ If you are defining parameters in the MDU, specify a data type of handle. The length will default to 0.
- ▶ If you are attributing your code, set the type property in the *xfParameter* attribute to *SynType.handle*. See the type property on [page 2-13](#).
- ▶ On a Java, COM, or .NET client, the parameter is handled as a String (or StringBuffer), BSTR, or string, respectively. See the data type mapping tables in [Appendix B](#) for details. If you are using the *xfNetLink* Java classes directly (rather than using class wrappers), see [synHandle on page 12-44](#).
- ▶ On a Synergy client, the argument is handled as a memory handle. Consequently, you must use the *RCB_xxx* routines to make remote calls. You cannot use *%RXSUBR*. When setting the arguments in the routine call block with either *RCB_SETARG* or *RCB_INSARG*, use the *D_TYPE_HANDLE* define and pass as the *data* argument the memory handle where the data is stored on the client. Returned data will be placed back into the same memory area. For details on using the *RCB_xxx* routines, see the “[Synergy Routine Call Block API](#)” chapter of the *Synergy Language Reference Manual*.
- ▶ If you are using *xfNetLink* Synergy and the memory area allocated to the handle is resized within the Synergy routine on the server, it will be resized accordingly on the client upon returning from the routine. Trailing blanks will be trimmed, so the resized memory area on the client may be smaller than it was on the server.

Passing a System.String Parameter

When using a .NET client, you can use a Synergy *System.String* to pass a parameter that is of variable length and/or larger than 64K. This method is an alternative to using a memory handle, as described above. *System.String* can also be used as a function return value.

Note the following:

- ▶ In your Synergy server routine, the parameter is declared as *@System.String*. For more information on using string data in Synergy Language see “[Data Types](#)” in the “Defining Data” chapter of the *Synergy Language Reference Manual* and “[System.String](#)” in the “System-Supplied Subroutines, Functions, and Classes” chapter of that same manual.

Preparing Your Synergy Server Code

Handling Variable-Length and Large Data

- ▶ If you are defining parameters in the MDU, specify a data type of `System.String`. The length will default to 0.
- ▶ If you are attributing your code, the data type will be obtained from your Synergy code.
- ▶ In your client code, instantiate a string. Refer to your .NET documentation for more information on .NET data types.

Returning a Collection of Structures

When using a Java or .NET client, you can return a structure collection parameter to the client from Synergy. A *structure collection* is an array of structures with a variable number of elements. The structure collection can only be used to send data *from* Synergy *to* the client. For a Java client, structure collections are supported only when using class wrappers.

On the client side, the structure collection is handled as an `ArrayList`; on the *xfServerPlus* side, the data is placed in a memory area. When the call returns from *xfServerPlus*, the `ArrayList` will be filled with structures and will know its own size (number of elements).

Note the following:

- ▶ Your Synergy server routine must declare the argument that will return the structure collection as a memory handle (**i4**; do not use **int**). *xfServerPlus* will create an empty memory area and pass the memory handle allocated to that area to your Synergy server routine. (You *must* use the memory handle provided by *xfServerPlus*; do not attempt to allocate your own.) Your routine must resize the area and place the structures in it. After the data has been sent to *xfNetLink*, *xfServerPlus* will free the memory area.
- ▶ If you are defining parameters in the MDU, specify a data type of structure, select the structure by name, and then select the Structure collection checkbox. The Data passed field will automatically be set to Out.
- ▶ If you are attributing your code, in the `xfParameter` attribute, set the `collectionType` property to `xfCollectType.structure` and specify the structure name with the structure property. See the `collectionType` property on [page 2-15](#).



For a .NET client, you can choose to have the ArrayList on the client created as a DataTable by selecting the DataTable checkbox in the MDU or setting the dataTable property of the xfParameter attribute to "true" if you are attributing your code. See ["Using DataTables" on page 17-18](#) for more information.

- ▶ On your Java or .NET client, instantiate an empty ArrayList (if there is any data in the list, it will be cleared) and pass it to the Synergy method that will return the structure collection. When the structure collection parameter is returned, use the ArrayList.size() method in Java or the ArrayList.Count property in .NET to get the size. Then, you can enumerate through the ArrayList to access the structures or access them by position (index). Refer to your Java or .NET documentation for more information on ArrayLists.

If you have defined a structure collection parameter in the SMC for use with your Java or .NET client, and you want to also use a Synergy or COM client with your server-side code, you can do the following:

- ▶ To use a Synergy client when your Java or .NET client passes a structure collection parameter, write your Synergy client code as though you were passing variable length data. That is, create a memory area on the client and pass the memory handle (as the parameter that is defined as a structure collection in the SMC) to xjServerPlus using the RCB_xxx routines. (See ["Passing a Single Parameter as a Memory Handle" on page 1-12.](#))

You do not need to alter the method definition in the SMC. When you make the call from your Synergy client, xjServerPlus will not give a "parameter mismatch" error because it has been programmed to permit a mismatch when a Synergy client passes a memory handle to a parameter that is defined as a structure collection in the SMC.

- ▶ To use a COM client when your Java or .NET client passes a structure collection parameter, you will need to write a separate Synergy routine that passes an array of structures and use that routine instead of the one that passes a structure collection. You may want to put this method (and any others that are specifically for COM) in a separate interface that you can include when building a type library, but exclude when building a JAR file or assembly.

Another benefit to creating a separate interface for COM-specific methods is that it allows you to give them the same method names as the non-COM methods.

When you build your type library, if you include the interface that contains the method with the structure collection, **gentlb** will omit that method when it creates the type library and issue a warning message regarding the omission.

Passing a System.Collections.ArrayList Parameter

When using a .NET client, you can use a Synergy System.Collections.ArrayList class to pass an ArrayList parameter either *from* Synergy *to* the client or *from* the client *to* Synergy. This method is an alternative to the one described in [“Returning a Collection of Structures” on page 1-14](#). In addition to allowing data to be passed in either direction, the other advantage to this method is that the elements in the ArrayList are not limited to structures. You may also pass data defined as alpha, decimal, implied-decimal, integer, or System.String data types. The ArrayList may vary in the number of elements it contains.

Note the following:

- ▶ In your Synergy server routine, declare the parameter as `@System.Collections.ArrayList`. See [“System.Collections.ArrayList”](#) in the “System-Supplied Subroutines, Functions, and Classes” chapter of the *Synergy Language Reference Manual* for details.
- ▶ If you are defining parameters in the MDU, in the Data type field, select the data type of the elements in the ArrayList. If the data type is structure, select the structure by name. Select the ArrayList checkbox and set Data passed to either In or Out. (In/Out is not supported.)
- ▶ If you are attributing your code, set the `collectionType` property of the `xfParameter` attribute to the data type of the array elements. If the data type is structure, you must also specify the structure name with the `structure` property. See the `collectionType` property on [page 2-15](#).



If the ArrayList contains structures, you can choose to have it created as a DataTable on the client by selecting the DataTable checkbox in the MDU or setting the `dataTable` property to “true” if you are attributing your code. See [“Using DataTables” on page 17-18](#) for more information.

- ▶ In your client code, instantiate an ArrayList. Note that although .NET ArrayLists support mixed data types, an ArrayList used with *xfNetLink* .NET may contain only one data type because mixed type ArrayLists are not supported in Synergy Language. Refer to your .NET documentation for more information on ArrayLists.

Passing Arrays Larger Than 64K

You can pass an array parameter larger than 64K (65,535 bytes) as long as no element in the array exceeds 64K. The maximum size for an arrayed field in Synergy Language is 256 MB.

Note the following:

- ▶ Your Synergy server routine should declare the argument that receives the array in the normal manner (that is, as an array argument of a particular data type).
- ▶ On a Java, COM, or .NET client, you do not need to do anything special on the client side to pass large arrays.
- ▶ On a Synergy client, the array must be placed in a memory handle, and you must use the RCB_xxx routines to make the remote call. You cannot use %RXSUBR. When setting the arguments in the routine call block with either RCB_SETARG or RCB_INSARG, you will use the D_TYPE_MEMARG define and pass as the *data* argument the memory handle where the array is stored on the client. Returned data will be placed back into the same memory area. For details on using the RCB_xxx routines, see the “[Synergy Routine Call Block API](#)” chapter of the *Synergy Language Reference Manual*.

Passing Binary Data

You can pass binary data, such as JPEG files, when using an *xfNetLink* Java or *xfNetLink* .NET client. On the client side, the binary data is handled as an *ArrayList* for a Java client and as a byte array for a .NET client.

In the MDU, define the parameter as a “Binary (handle)” data type. (If you are attributing your code, set the type property of the *xfParameter* attribute to *SynType.binaryhandle*; see [page 2-13](#).) Your Synergy server routine must declare the argument that receives the data as a memory handle (**i4**; do not use **int**). *xfServerPlus* will place the data in a memory area and pass the memory handle allocated to that area to your Synergy server routine. (You *must* use the memory handle provided by *xfServerPlus*; do not attempt to allocate your own.) After the data has been returned to *xfNetLink*, *xfServerPlus* will free the memory area.



If you are using a .NET client, binary fields in structures are converted to byte arrays by default. However, you should use the procedure described in this section to pass binary data such as JPEG files rather than a binary field in a structure, because the latter requires that you specify a size. See also the description of the **gencs -nb** option on [page 16-18](#).

For details on passing binary data, including code samples, refer to the following:

- ▶ Java, see [page 10-17](#)
- ▶ .NET, see [page 17-22](#)

If you have defined a binary (handle) parameter in the SMC for use with your Java or .NET client, and you want to also use an *xfNetLink* Synergy or *xfNetLink* COM client with your server-side code, take note of the following:

- ▶ To use an *xfNetLink* Synergy client when your Java or .NET client passes a binary (handle) parameter, write your Synergy client code as though you were passing variable length data. That is, create a memory area on the client and pass the memory handle (as the parameter that is defined as a binary (handle)) to *xfServerPlus* using the *RCB_xxx* routines. (See “[Passing a Single Parameter as a Memory Handle](#)” on [page 1-12](#).)

You do not need to alter the method definition in the SMC. When you make the call from your Synergy client, *xfServerPlus* will not give a “parameter mismatch” error because it has been programmed to permit a mismatch when

a Synergy client passes a memory handle to a parameter that is defined as a binary (handle) parameter in the SMC.

- There is no workaround for using a COM client when your Java or .NET client passes a binary (handle) parameter. When you create the type library, **gentlb** will omit the method with the binary (handle) parameter and issue a warning message regarding the omission. (Note that you cannot use a handle parameter as a workaround for passing binary data on COM because it passes a string. The binary data may contain nulls, which would cause the string to terminate.)

Handling Errors

A distributed computer system has multiple points of failure, making it complex to debug. Because the user interface is on the client machine, errors in your server application cannot be handled by writing a message to the screen of the machine on which your Synergy routines are executed. (This could be the screen of a server that no one is looking at.) Nor can you rely on information from tracebacks. You *must* trap and handle errors in some other manner. The best way to do this is to trap for all possible errors on the server side and have your routines return status information (success, failure, or an error number) via the return result or the argument list. This allows processing to continue even though an error occurred. On the client side, your code should check for errors—both *xfServerPlus* errors and errors from the client code.

For more information on error handling in modular code, see the Synergex white paper, *Modularizing Your Synergy Code: The First Step to Distributed Computing*, available on the Online Manuals CD. For information on handling errors for the individual *xfNetLink* clients, refer to the following:

- Synergy, see [page 6-3](#)
- Java, see [page 11-1](#)
- COM, see [page 15-1](#)
- .NET, see [page 18-1](#)

Testing Your Synergy Code

Once your code is modularized and contained in ELBs, you should test it thoroughly in a stand-alone configuration. It will be easier to troubleshoot and run the debugger now, than it will be when there is a client and a remote connection involved. When you are sure that your code is working locally, we recommend that you test it with a remote connection using an *x/f*NetLink Synergy client before adding a Java, COM, or .NET client to the picture.

To make it easier to test your code, we've included a test skeleton generator utility (**gensyn.dbr**), which can be run from Workbench or from the command line. This utility generates skeletal code from your Synergy Method Catalog (SMC) definitions, which you can edit to create a test program. You can choose to create test skeletons that contain local, remote, or debug calls to the Synergy routines you have modularized and contained in an ELB. The local calls are made with XSUBR; the remote and debug calls use the routines in the *x/f*NetLink Synergy API.

Before generating test skeletons, your code should be modularized and contained in ELBs, and you must define your Synergy routines in the Synergy Method Catalog. See [chapter 2](#) for information about the SMC.



You cannot generate test skeletons if your Synergy routines defined in the SMC include any of the following:

- ▶ parameters with a data type of handle, binary (handle), System.String, or enumeration
 - ▶ parameters that are flagged as structure collections or ArrayLists
 - ▶ routines with a return value of System.String or enumeration
-

The generated code is not a complete, ready-to-run, test program. For example, you will have to edit the generated code to ensure that calls are made in the proper sequence and that parameters are initialized. Items that require your attention are marked with “TO_DO” in the code. Once the TO_DOs are completed, you can compile, link, and run the test programs.

The test skeletons are intended primarily for developers who want to test their Synergy code before building a component with *x/f*NetLink Java, *x/f*NetLink COM, or *x/f*NetLink .NET. If you are using *x/f*NetLink Synergy, you can still generate test skeletons, as long as you complete the Interface name field when defining routines in the MDU.

Generating Test Skeletons with Workbench



The Generate Synergy Test Skeletons menu entry is not included in new projects starting with version 9.3. The entry will still appear on the Build menu in older projects. You can access the dialog in any Synergy project by typing `SynStartSkeletonGen` on the Workbench command line, or you can generate test skeletons from the command line (see [page 1-23](#)). If you use this feature frequently, you may want to add it as a menu option (using Project > Project Properties > Tools) in Workbench.

1. From within the Synergy/DE project that contains your modularized code, select Build > Generate Synergy Test Skeletons.



You can also generate test skeletons by selecting Build > Generate Synergy Test Skeletons in a Synergy/DE Java, COM, or .NET Component project.

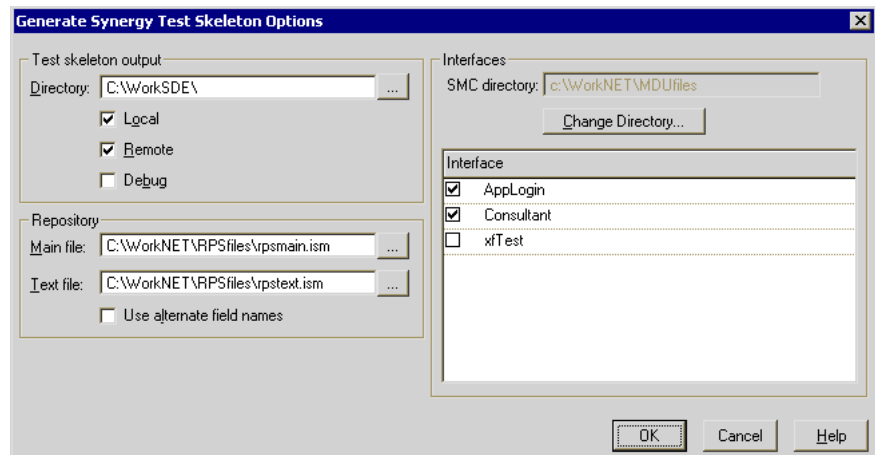


Figure 1-1. The Generate Synergy Test Skeletons Options dialog box.

2. Complete the following fields in the Generate Synergy Test Skeleton Options dialog box:

Directory. Specify the directory to which you want the test skeleton files saved. If you enter a logical in this field, it must be followed by a colon (e.g., DBLDIR:). The default is the directory where the project is stored.

Preparing Your Synergy Server Code

Testing Your Synergy Code

Local. Select Local if you want to generate a test skeleton that makes local calls to the routines in your ELBs.

Remote. Select Remote if you want to generate a test skeleton that uses *x/NetLink Synergy* and *x/ServerPlus* to make remote calls to the routines in your ELBs.

Debug. Select Debug if you encountered errors while running the remote test program and want to run your *x/ServerPlus* session in debug mode. This option is applicable only if the operating system of your *x/ServerPlus* machine is Windows or UNIX. For OpenVMS, see the note on [page 1-28](#). To perform remote debugging via Telnet, you do not need a debug test skeleton; use the remote skeleton instead. (See “[Debugging Your Remote Synergy Routines](#)” on [page 3-40](#) for information about the Telnet method.)

Repository main file. (optional) If any of the methods that will be included in the test skeletons pass structures as parameters, specify the location of the repository main file for those structures. This must match the repository that was used when entering data in the SMC. If you specify a main file, you must also specify a text file.

The default is the value of the environment variable RPSMFIL. If it is not defined, the default is **RPSDAT:rpsmain.ism**. If neither RPSMFIL nor RPSDAT are defined, the default is **rpsmain.ism** in the path specified in the Working directory property of the project. If the Working directory is not defined, the default is **rpsmain.ism** in the location where the project is stored.

Repository text file. (optional) If any of the methods that will be included in the test skeletons pass structures as parameters, specify the location of the repository text file for those structures. This must match the repository that was used when entering data in the SMC. If you specify a text file, you must also specify a main file.

The default is the value of the environment variable RPSTFIL. If it is not defined, the default is **RPSDAT:rpstext.ism**. If neither RPSTFIL nor RPSDAT are defined, the default is **rpstext.ism** in the path specified in the Working directory property of the project. If the Working directory is not defined, the default is **rpstext.ism** in the location where the project is stored.

Use alternate field names. Check this box if you want the field names in your structures to use the value in the Alternate name field in Repository instead of the value in the Name field.

SMC directory. This field displays the path for the Synergy Method Catalog that will be used to generate the code for the test skeletons. The default is DBLDIR. To change the SMC directory, click the Change Directory button to display the Browse for SMC Directory dialog box. Navigate to the directory, double-click to select it, and click OK. The selected path will display in the SMC directory field and the list of interfaces will be refreshed, displaying all interfaces in the selected SMC.

Interfaces. Select the interfaces for which you want to generate test skeletons by clicking in the box to the left of the interface name. The test skeleton utility will create a DBL file of each type you requested (local, remote, debug) for each interface. The DBL files are named with the interface name plus the type of skeleton. For example, if the interface is named “Consultant”, the local test skeleton will be named **Consultant_local.dbl**.

3. Click OK to generate the test skeletons.
4. Open the .dbl files and edit the TO_DO items. See [“Using the Test Skeletons” on page 1-25](#) for specifics on what you need to edit, as well as information on compiling, linking, and running the three types of test programs.

Generating Test Skeletons from the Command Line

Generating test skeletons from the command line is a two-step process. First, you will run **genxml** to create a Synergy XML file. Then, you’ll run **gensyn** to create the test skeletons. Both **genxml** and **gensyn** are located in DBLDIR and run on all supported Windows, UNIX, and OpenVMS platforms.

1. Run **genxml** with the **-f** and **-i** options. You may also need to use the **-d**, **-s**, and **-n** options. See [“The genxml Utility” on page 9-9](#), [page 13-11](#), or [page 16-13](#) for detailed information on the **genxml** syntax. For example, on Windows:

```
dbr DBLDIR:genxml -f temp -i Consultant -d c:\work  
-s c:\work\SMC
```

This will use the SMC located in the c:\work\SMC directory to create a Synergy XML file named **temp.xml**, which contains information about the Consultant interface. The XML file will be placed in the c:\work directory.

2. Run **gensyn**, passing the name of the XML file output by **genxml**. See [“The gensyn Utility” on page 1-24](#) for the complete syntax. For example, on Windows:

```
dbr DBLDIR:gensyn -f c:\work\temp.xml -d c:\work -l -r
```

Preparing Your Synergy Server Code

Testing Your Synergy Code

Continuing our example from above, this command will produce two files—**Consultant_local.dbl** and **Consultant_remote.dbl**—and place them in the `c:\work` directory.

3. Open the **.dbl** files and edit the **TO_DO** items. See [“Using the Test Skeletons” on page 1-25](#) for specifics on what you need to edit, as well as information on compiling, linking, and running the three types of test programs.

The gensyn Utility

The **gensyn** utility creates test skeletons from an XML file created by **genxml**.

Syntax `dbr gensyn -f xmlFilename [-d targetDir] [-l] [-r] [-b]
[-v msgLevel] [-?]`

Arguments `-f xmlFilename`

The name of the XML file generated by **genxml**. Include the full path if necessary; if the path is not included, the current directory is assumed.

`-d targetDir`

(optional) The directory for the DBL files. This must be the full path of an existing directory; do not use logicals. If not specified, the files are created in the current directory.

`-l`

(optional) Generate test skeletons that make local calls to the routines in your ELBs.

If no test skeleton type is specified with **-l**, **-r**, or **-b**, all three types will be generated.

`-r`

(optional) Generate test skeletons that use *xfNetLink* Synergy and *xfServerPlus* to make remote calls to the routines in your ELBs.

`-b`

(optional) Generate debug test skeletons. Debug test skeletons are useful if you encountered errors while running the remote test program and want to run your *xfServerPlus* session in debug mode. This option is applicable only if the operating system of your *xfServerPlus* machine is Windows or UNIX. For OpenVMS, see the note on [page 1-28](#). To perform remote debugging via

Telnet, you do not need a debug test skeleton; use the remote skeleton instead. See “[Debugging Your Remote Synergy Routines](#)” on page 3-40 for information about the Telnet method.

-v *msgLevel*

(optional) Level of verbosity in messages.

0 = no messages

1 = error messages

2 = error messages and the names of generated files (default)

-?

(optional) Displays a list of options and the version number for **gensyn**.

Using the Test Skeletons

The generated code includes a subroutine for each method in the selected interface. For each subroutine there is a local record structure, which defines the parameters passed by the method.

Using the local test skeleton

The local test skeleton makes calls using XSUBR.

1. Complete the TO_DO items in the *_local.dbl file.
 - Put the method calls in the correct sequence. You may also want to comment out some calls, depending upon the requirements of your application. For example, in some applications, it may not be valid to make all calls in sequence. Instead, you may need to test the method calls a few at a time.
 - Supply dimensions if any of your methods pass array parameters.
 - Initialize input parameters with valid data. In the test skeleton, input parameters and structure field values are initialized to zero or blank. You should initialize any parameters for which the Synergy routine expects to receive a value. Structure parameters are treated as alpha strings in the test skeleton. This means that integer data in structures will be considered alpha data and will be passed “as is”.



If you regenerate the test skeleton, the changes you have made to the **.dbl** file will be lost.

Preparing Your Synergy Server Code

Testing Your Synergy Code

2. Compile and link `*_local.dbl`. You should not link against your ELB because the code in `*_local.dbl` includes a call to `OPENELB`.
3. Run `*_local.dbr`. The program will call `OPENELB` and then call each routine in the interface.

If you encounter errors, there is likely a problem in the modularized Synergy routines. Correct the code, recompile it into an ELB, and run the local test program again.

For errors that require changes to the SMC (e.g., a missing parameter), you will need to correct the SMC and then either regenerate the test skeleton or modify the test program by hand. When you regenerate the test skeletons, you will lose the edits you made. So, depending on the extent of the changes, it may be more efficient to modify the program by hand than to regenerate.

Using the remote test skeleton

The remote test skeleton code makes calls using the *x/NetLink* Synergy API. The API, along with information on setting up your system, making remote calls, and troubleshooting *x/NetLink* Synergy is documented in [Part II](#) of this manual.

1. Edit the `TO_DO` items in `*_remote.dbl`.
 - ▶ Modify the IP define statement to be the IP address or host name of your *x/ServerPlus* machine. (It defaults to the name of the machine on which the test skeletons were generated.)
 - ▶ Modify the PORT define to be the port on which *x/ServerPlus* is running.
 - ▶ Put the method calls in the correct sequence. You may also want to comment out some calls, depending upon the requirements of your application. For example, in some applications, it may not be valid to make all calls in sequence. Instead, you may need to test the method calls a few at a time.
 - ▶ Supply dimensions if any of your methods pass array parameters.
 - ▶ Initialize input parameters with valid data. In the test skeleton, input parameters and structure field values are initialized to zero or blank. You should initialize any parameters for which the Synergy routine expects to

receive a value. Structure parameters are treated as alpha strings in the test skeleton. This means that integer data in structures will be considered alpha data and will be passed “as is”.



If you regenerate the test skeleton, the changes you have made to the **.dbl** file will be lost.

2. Compile and link *_remote.dbl.

This is the client part of the application, which will make remote calls to the ELBs on your *xfServerPlus* machine. Note that you do not have to use a separate machine for the client; if you do, however, Core Components (Windows) or Synergy Language (UNIX and OpenVMS) must be installed on the client to run *xfNetLink Synergy*.

3. Verify that *xfServerPlus* is installed, configured, and running on your Synergy server machine. We recommend that you turn logging on while running the test programs. See [chapter 3](#) for information on configuring and running *xfServerPlus*.
4. Verify that your SMC files and ELBs are on the Synergy server machine.
5. Run *_remote.dbr on the client. The program will start an *xfServerPlus* session, call each of the methods in the interface, and then shut down the session.

If you encounter errors while running the remote program, check the table on [page 7-20](#) for status codes returned during session start-up or the table on [page 7-26](#) for errors signaled by the %RXSUBR call.

If the problem is in your Synergy code, correct it and re-run the remote test program. If the problem is in the method definitions in the SMC, you must correct the SMC and then either regenerate and re-edit the test skeleton or modify the test skeleton by hand before retesting. Depending on the extent of the changes, it may be more efficient to modify the program by hand than to regenerate and re-edit the test skeleton.

If the problem appears to be related to the setup of *xfServerPlus*, you may want to run the *xfNetLink Synergy* test program and the *xfServerPlus* test program. See “Testing *xfNetLink Synergy*” on [page 5-6](#) and “Testing *xfServerPlus*” on [page 3-15](#).

Using the debug test skeleton

The debug test skeleton makes calls using the *xfNetLink* Synergy API. It enables you to manually connect an *xfServerPlus* session to your client so that you can run the debugger on the Synergy code in your ELBs. See [“Running an *xfServerPlus* Session in Debug Mode” on page 6-5](#) for detailed information.



If the operating system of your *xfServerPlus* machine is OpenVMS, you do not need to use the debug test skeleton to run in debug mode. Instead, use the remote skeleton. See [“Running in debug mode on OpenVMS” on page 6-7](#).

1. Edit the TO_DO items in `*_debug.dbl`.
 - ▶ Put the method calls in the correct sequence. You may also want to comment out some calls, depending upon the requirements of your application. For example, in some applications, it may not be valid to make all calls in sequence. Instead, you may need to test the method calls a few at a time.
 - ▶ Supply dimensions if any of your methods pass array parameters.
 - ▶ Initialize input parameters with valid data. In the test skeleton, input parameters and structure field values are initialized to zero or blank. You should initialize any parameters for which the Synergy routine expects to receive a value. Structure parameters are treated as alpha strings in the test skeleton. This means that integer data in structures will be considered alpha data and will be passed “as is”.



If you regenerate the test skeleton, the changes you have made to the `.dbl` file will be lost.

2. Compile and link `*_debug.dbl`.

This is the client part of the application, which will make remote calls to the ELBs on your *xfServerPlus* machine.
3. Verify that *xfServerPlus* is installed, configured, and running on your Synergy server machine. We recommend that you turn logging on while running the test programs. See [chapter 3](#) for information on configuring and running *xfServerPlus*.
4. Verify that your SMC files and ELBs are on the Synergy server machine.

5. Run `*_debug.dbr`. The program will start a connection to *xfServerPlus* with `%RX_DEBUG_INIT` and display the IP and port for listening. You will then need to manually start the *xfServerPlus* session in debug mode. See the instructions in [“Running an xfServerPlus Session in Debug Mode” on page 6-5](#).

If you encounter errors while running the debug program, check the table on [page 7-20](#) for status codes returned during session start-up or the table on [page 7-26](#) for errors signaled by the `%RXSUBR` call.

Preparing Your Synergy Server Code

Using the xfServerPlus Application Program Interface

Using the *xf*ServerPlus Application Program Interface

Three *xf*ServerPlus API subroutines, SET_XFPL_TIMEOUT, XFPL_LOG, and XFPL_REGCLEANUP, are available for use in your Synergy applications. These subroutines are stored in **xfpl_api.elb** (XFPL_API.EXE on OpenVMS) in DBLDIR.

SET_XFPL_TIMEOUT

```
xcall SET_XFPL_TIMEOUT(minutes[, seconds])
```

This subroutine sets a time-out value for the remote execution server in *xfServerPlus*. The time-out period starts over after *each* call–confirm cycle.

Arguments *minutes*

The number of minutes to wait before shutting down *xfServerPlus*. (n)

seconds

(optional) The number of seconds to wait before shutting down *xfServerPlus*. (n)

Discussion SET_XFPL_TIMEOUT can be used to handle cases where the client exits or shuts down abnormally and *xfServerPlus* does not receive a shutdown message. For example, when you are developing or testing, it may be helpful to set a small time-out value so that open ELBs are closed promptly and can be updated. (The ELBs and shared images that have been opened by *xfServerPlus* are closed when it shuts down.)

We do not recommend using SET_XFPL_TIMEOUT if you are using pooling because it can cause connections in the pool to time out. We recommend using the client-side time-out options instead (see below).

Ideally, your client should always send a shutdown message to *xfServerPlus*. However, if no shutdown message is sent, and no time-out is specified with SET_XFPL_TIMEOUT, *xfServerPlus* may continue running until the machine on which it resides is shut down or the KEEPALIVE timer value is reached and the daemon shuts down the connection. (For more information on the KEEPALIVE timer, see the *Installation Configuration Guide* or your operating system documentation.) As long as *xfServerPlus* continues running, one of your *xfServerPlus* sessions is in use, tying up one of your licenses.

If SET_XFPL_TIMEOUT is called and XFPL_SESS_INFO is set to “all” in the *xfpl.ini* file, an entry is made in the *xfServerPlus* log file (*xfpl.log*), recording the time-out value (assuming logging is turned on, of course). If *xfServerPlus* times out, that fact will be recorded in the log. The error you will see in your program depends on the state of the client application when it attempts to access the timed-out connection. Your client application should be prepared to handle this.

Preparing Your Synergy Server Code

SET_XFPL_TIMEOUT

You can also set time-out values for several operations in the *x/NetLink* clients:

- ▶ Synergy, see [“Specifying Time-out Values” on page 5-3](#)
- ▶ Java, see [“Specifying Time-out Values” on page 10-8](#)
- ▶ COM, see [“Setting a Call Time-Out” on page 14-12](#)
- ▶ .NET, see [“Setting a Call Time-Out” on page 17-24](#)

XFPL_LOG

```
xcall XFPL_LOG(text_string)
```

This subroutine makes an application-defined entry in the *xfServerPlus* log file.

Arguments *text_string*

The text that you want to write to the log. (a)

Discussion To use this routine, you must activate logging in the **xfpl.ini** file by setting XFPL_LOG to ON. See [“Using Server-Side Logging” on page 3-29](#) for more information.

The Synergy Method Catalog is pre-loaded with an entry for XFPL_LOG, so that it can be called from the client. If this entry is missing, you can import it from the **defaultsmc.xml** file. See [“Importing and Exporting Methods” on page 2-39](#).

XFPL_LOG must be included in the SMC if you are using the `setUserString()` methods in *xfNetLink Java*, *xfNetLink COM*, or *xfNetLink .NET*.

This routine opens and closes a channel. If you get a “Channel is in use” error when using XFPL_LOG, see [“Specifying a Base Channel Number” on page 1-7](#). If you are using U_START, also see the discussion on [page 1-6](#).

XFPL_REGCLEANUP

```
xcall XFPL_REGCLEANUP(methodID)
```

This subroutine registers a cleanup method that will be run automatically on the server when *xfServerPlus* unexpectedly loses socket communication with the client. On OpenVMS, the cleanup method also runs when communication with the **rsynd** process is lost. The cleanup method is called by *xfServerPlus* *after* the remote execution server has timed out. You can set a time-out value for *xfServerPlus* with SET_XFPL_TIMEOUT (see [page 1-31](#)).

Arguments *methodID*

The method ID (as entered in the SMC) of the cleanup method to run. The method ID is case sensitive. (a)

Discussion XFPL_REGCLEANUP can be used with any *xfNetLink* client following the steps in “[To use XFPL_REGCLEANUP](#)” below. If you have an *xfNetLink* Java, *xfNetLink* COM, or *xfNetLink* .NET client and are using the pooling support methods, XFPL_REGCLEANUP and the corresponding cleanup method work differently; see “[To use XFPL_REGCLEANUP with the pooling support methods](#)” on [page 1-35](#).

The default SMC is pre-loaded with an entry for XFPL_REGCLEANUP. If this entry is missing, you can import it from the **defaultsmc.xml** file. See “[Importing and Exporting Methods](#)” on [page 2-39](#).

► To use XFPL_REGCLEANUP

Follow these instructions to use XFPL_REGCLEANUP when you are using a Synergy client or are *not* using the pooling support methods with a Java, COM, or .NET client.

xfServerPlus calls the cleanup method registered by XFPL_REGCLEANUP only when there is a failure that causes socket communication to be lost. *xfServerPlus* does not call this cleanup method when a session is ended in the normal manner. If you want to perform any cleanup tasks during normal session shutdown, you will need to explicitly call either this cleanup method or a separate cleanup method, depending on the needs of your application.

1. Run the MDU to verify that there is an entry for XFPL_REGCLEANUP in the Synergy Method Catalog. If you're creating a Synergy component, assign the XFPL_REGCLEANUP routine an interface name, and then include that interface when you create your component.
2. Write a Synergy routine to perform the cleanup. Cleanup may include closing or releasing files, writing to a log, and so forth. The cleanup routine must be a subroutine (that is, it cannot have a return value) and must have no parameters.
3. Use the MDU to add your cleanup routine to the Synergy Method Catalog.

If you are creating a Synergy component and the cleanup method will be used only with XFPL_REGCLEANUP, you do not have to specify an interface name for the cleanup method because you do not need to include it in your component. If you will call this same cleanup method from your client program to do cleanup on a normal shutdown, you will need to specify an interface name for it and then include it in your component.

4. Call XFPL_REGCLEANUP from your client program and pass the method ID of the cleanup method. You should do this right after starting a session, so that the cleanup method is registered on the server and ready to use should it be needed. If XFPL_REGCLEANUP is called more than once, the most recently registered method will be called when socket communication is lost.



You can also call XFPL_REGCLEANUP from your server code after the session has started. If you plan to call XFPL_REGCLEANUP only from the server code, it does not require an entry in the SMC, so you can skip step 1.

► To use XFPL_REGCLEANUP with the pooling support methods

These instructions apply only when you are using the pooling support methods with a Java, COM, or .NET client. These methods are available with connection pooling on *xy*NetLink Java and with object pooling on *xy*NetLink COM and *xy*NetLink .NET. (For general information about pooling, see [“Understanding Java Connection Pooling”](#) on page 10-20, [“Understanding COM Pooling”](#) on page 14-14, or [“Understanding .NET Pooling”](#) on page 17-26.)

When you are using pooling, the cleanup method that you write is called both when a session ends abnormally because socket communication has been lost and when a session ends normally.

Preparing Your Synergy Server Code

XFPL_REGCLEANUP

1. Run the MDU to verify that there is an entry for XFPL_REGCLEANUP in the Synergy Method Catalog. Because XFPL_REGCLEANUP is called automatically when the pool is created, you do not need to assign it an interface name and include it in your Synergy component.
2. Write a Synergy routine to perform the cleanup tasks. For details on writing and using a cleanup method, refer to the relevant sections for your client.
 - ▶ Java, see [“Using the Pooling Support Methods” on page 10-32](#) and [“Cleanup method” on page 10-34](#)
 - ▶ COM, see [“Using the Pooling Support Methods” on page 14-22](#) and [“Cleanup method” on page 14-24](#)
 - ▶ .NET, see [“Using the Pooling Support Methods” on page 17-32](#) and [“Cleanup method” on page 17-34](#)
3. Use the MDU to add your cleanup routine to the Synergy Method Catalog.
 - ▶ For COM and .NET, the method name must be “Cleanup” (case sensitive). Add the cleanup routine to the interface for the object it applies to so that it can be included when you build your TLB or assembly.
 - ▶ For Java, do not put the cleanup routine in an interface. You do not need to include it in your JAR file, and there are no restrictions on how it should be named.

Defining Your Synergy Methods

The Synergy Method Catalog (SMC) identifies the Synergy functions and subroutines that you have prepared for remote calling. These functions and subroutines are referred to as *Synergy methods*. The SMC identifies each Synergy method, the interface it is associated with, its subroutine or function name, and the ELB or shared image in which it can be found. The SMC also provides detailed information on parameters and function return values.

The SMC consists of the files `cdt.is?` and `cmpdt.is?`. You cannot change these filenames; however, you can create additional SMCs in different directories if desired. (See “[Defining Multiple Synergy Method Catalogs](#)” on page 2-43.) The SMC must be located on the machine running *xfServerPlus*. A default SMC is installed with *xfServerPlus* in the DBLDIR directory. It contains methods used by the test programs, as well as entries for `XFPL_LOG` and `XFPL_REGCLEANUP`.

The SMC data is used by *xfServerPlus* to

- ▶ allocate adequate memory for data that is passed to and updated by Synergy Language routines.
- ▶ ensure that data from *xfNetLink* clients is translated into the correct Synergy Language data types.
- ▶ ensure that data returned to the clients is translated into the correct *xfNetLink* data types.

Data in the SMC is also used to create the following:

- ▶ JAR files for use with *xfNetLink* Java
- ▶ type libraries for use with *xfNetLink* COM
- ▶ assemblies for use with *xfNetLink* .NET
- ▶ test skeletons used to test your Synergy server code

Defining Your Synergy Methods

Understanding Routine Name, Method Name, and Method ID

Populating the SMC

There are two ways to populate the SMC with data about your routines:

- ▶ Load data directly into the SMC from an XML file generated by **dbl2xml**. The **dbl2xml** utility reads information about your routines from your Synergy source code and generates an XML file, which you can import into the SMC. See [“Using Attributes to Define Synergy Methods” on page 2-3](#).
- ▶ Use the Method Definition Utility (MDU), which provides a graphical user interface for data entry. See [“Using the MDU to Define Synergy Methods” on page 2-22](#).

Understanding Routine Name, Method Name, and Method ID

Whether you load data from an XML file or use the MDU to populate the SMC, you will be dealing with the routine name, method name, and perhaps method ID. The purpose of this section is to explain what the routine name, method name, and method ID are and how they have evolved over time, and also to offer some suggestions on how best to use these fields depending on which *xjNetLink* client you are using.

The *routine name* is simply the name of your Synergy subroutine or function.

The *method name* (introduced in version 8.3) is a 50-character value that you create to reference the Synergy routine. It is used by *xjNetLink* Java (with class wrappers), *xjNetLink* COM, and *xjNetLink* .NET to invoke the Synergy routine within your client code. Methods are grouped into interfaces for inclusion in a Synergy component; consequently, the method name must be unique within an interface. When you generate a JAR file, type library, or assembly, the interface name becomes the class name, and the method name in the SMC is used as the name of the generated method within that class. Prior to 8.3, the routine name was used as the name of the generated method; adding a separate field for the method name provides much-needed flexibility, as it allows the method name in the client code to differ from the routine name on the server side.

The *method ID* is a unique, 31-character value that is used to identify the Synergy routine. *xjServerPlus* uses this value to look up the routine to call in the SMC; consequently, the method ID must be unique within the method catalog. Prior to 8.3, you had to explicitly specify a method ID; starting with version 8.3, the method ID is generated for you from the value in the Method name field. (This is

also the case when using attributes to define your methods: the method ID defaults from the method name.) You can change the generated method ID if desired. Users of *xfNetLink* Java (with class wrappers), *xfNetLink* COM, and *xfNetLink* .NET never directly reference the method ID in the client code. Users of *xfNetLink* Synergy and *xfNetLink* Java (without class wrappers) reference the method ID in the client code to invoke the associated Synergy routine. Although you can use the routine name as the method ID, there are advantages to using a different value for the method ID. If the Synergy routine name is cryptic, you can use a more meaningful name as the method ID. In addition, you can define the same routine multiple times in the SMC, in case your client program needs to call it in different ways.

Because *xfNetLink* Synergy and *xfNetLink* Java (without class wrappers) use the method ID to invoke the Synergy routine, they do not use the value in the Method name field. However, because the MDU copies the value you enter in the Method name field to the Method ID field, we recommend that when you are entering new methods, you enter the value you want to use for the method ID in the Method name field. Keep in mind that while the method name may be up to 50 characters long, the method ID is limited to 31 characters. If the method name is longer than 31 characters, it will be truncated when it is copied to the method ID.

Using Attributes to Define Synergy Methods

The recommended method for populating the SMC is to attribute your Synergy code, use the **dbl2xml** utility to produce an XML file, and then import definitions from the XML file into the SMC. The **dbl2xml** utility (installed with Synergy/DE Professional Series) parses information about the routines you want to use with *xfServerPlus* from your Synergy source code. The utility outputs an XML file, which is then imported into the SMC using the MDU's import facility. When using this method, you are not required to use the MDU to perform data entry.

The **dbl2xml** utility obtains much of the basic information about your routines, such as the routine name and return value data type, from the code itself; however, some types of information, such as the interface name, must be provided by adding attribute statements to your code. There are two attributes, **xfMethod** and **xfParameter**, each of which has a number of properties. **xfMethod** is required, but **xfParameter** may be optional, depending on the type of parameter you are

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

defining. The example below shows a simple `xfMethod` attribute statement for the function `ReturnError`. (See [“Attribute Examples” on page 2-16](#) for additional examples.)

```
{xfMethod(interface="ConsultApp", elb="EXE:Consult")}  
function ReturnError ,string  
    req in userToken          ,a22  
endparams
```

The advantage to attributing your code and using **dbl2xml** over using the MDU to perform data entry is that your code and SMC are less likely to get out of sync. You may, of course, need to alter the attributes and properties if you change your code, but because the attributes are right there in the source file, you are unlikely to forget to do so. Additionally, you can add the command to run the **dbl2xml** utility to a build script so that it runs every time you compile. You can also add the MDU command line option to update the SMC to your build script, thereby automating the SMC update process. Although there is some up-front work to implement the attributes, once you have done so, using attributes should prove more efficient and accurate than using the MDU to input and maintain data about your routines.

The **dbl2xml** utility also processes documentation comments in your Synergy code. These comments are imported along with your method definitions into the SMC and can be used to create API documentation for your Java or .NET component. See [“Documentation Comments” on page 2-19](#).

General Procedure

1. Modify your code to include attributes, parameter modifiers, and documentation comments. See the table on [page 2-6](#) for assistance in determining what additions or changes you need to make to your code. For example, you may need to add direction modifiers to your parameter definitions. The only required properties are `interface` and `elb` in the `xfMethod` attribute, but you may need to add others depending on your code and the desired results on the client side.



If you are attributing an existing `xfServerPlus-xfNetLink` application, pay special attention to how methods are named so that you don't break existing client code. See the table on [page 2-6](#) for the rules on defaulting.

2. Compile your code and fix any resulting problems.
3. Run the **dbl2xml** utility on your source files. It will create an XML file that contains interface definitions, and which is named (by default) with the first interface in the first source file. If desired, you can specify a name for the XML file with the **-out** option. For example:

```
dbl2xml VendorMaint.dbl -out c:\work\Vendor.xml
```

See “[The dbl2xml Utility](#)” in the “General Utilities” chapter of *Synergy Language Tools* for the complete **dbl2xml** syntax.



The XML generated by **dbl2xml** is very similar to that generated by **genxml**, but it is not exactly the same. The file generated by **dbl2xml** can be used only to update the SMC; it cannot be read by **genjava**, **gentlb**, or **gencs**.

4. Run the MDU from the command line with the **-u** option or open the MDU application and select Utilities > Import Methods to import the data from the XML into the SMC. For example,

```
dbf dbl_dir:mdu d:\synergy\smcFiles -u c:\work\Vendor.xml
```

5. In the future, when you make changes to your Synergy code, make the corresponding changes to the attributes (if necessary), and then repeat [step 2](#) through [step 4](#).



You may want to check your results in the MDU after importing the XML file to ensure methods and parameters are defined as you intended. Pay special attention if you rely on default sizes for return values and parameters, as the data type conversion for the client sometimes depends on the size defined in the SMC.

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

Routine metadata and where it comes from

The table below summarizes the information we need to know about the routines you want to use with *xyServerPlus* and indicates whether that information is obtained from source code, attributes, or default values. The applicable attributes and properties are included in the table; see the referenced pages for the exact syntax and details on usage.

Item	Source/Comments	Attribute	Property
Routine name	The function or subroutine name is obtained from the source code.	N/A	N/A
Method name	Defaults to the Synergy routine name, but can be overridden with the name property. See page 2-9 .	xfMethod	name="xxx"
Method ID	Defaults to the Synergy routine name or to the method name if the name property is used. Can be overridden with the id property. See page 2-9 .	xfMethod	id="xxx"
Interface name	Specified with the interface property. This property is required. See page 2-8 .	xfMethod	interface="xxx"
ELB name	Specified with the elb property. This property is required. See page 2-9 .	xfMethod	elb="xxx"
Return value data type	Obtained from the function definition or, if not specified there, defaults to the data type of the variable or literal of the first FRETURN statement. (In other words, the same way the compiler determines the return value data type.) To coerce the data type to a non-default type on the client, use the cType property; see page 2-10 . Return values that are coerced to DateTime data type require the format property; see page 2-12 . DateTime return values can be created as nullable types on the client using the nullable property; see page 2-12 .	xfMethod	cType=xfType.xxx format=xfFormat.xxx nullable=true
Return value size	Obtained from the function definition or, if not specified there, a default value is used for most data types; else, you must specify it with the length property (and precision property, if necessary). See page 2-10 .	xfMethod	length=xxx precision=xxx

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

Item	Source/Comments	Attribute	Property
Parameter name	Defaults to the declared Synergy parameter name, but can be overridden with the name property. See page 2-13 .	xfParameter	name="xxx"
Parameter data type	Obtained from the parameter definition for most data types. In some cases, must be specified with the type property. See page 2-13. To coerce the parameter to a non-default type on the client, use the cType property; see page 2-14. Parameters that are coerced to DateTime require the format property; see page 2-14. A DateTime parameter can be created as a nullable type on the client using the nullable property; see page 2-15. For structures passed as parameters or arrays, the data type is obtained from the parameter definition (which must be defined as a structfield). See example B on page 2-16 (parameter) and example J on page 2-19 (array). For structures passed as ArrayLists or structure collections, use the collectionType and structure properties. See page 2-15. To specify that an array or ArrayList of structures be created as a DataTable on the client, use the dataTable property. See page 2-16.	xfParameter	type=SynType.xxx cType=xfType.xxx format=xfFormat.xxx nullable=true collectionType=xfCollectType.xxx structure="xxx" dataTable=true
Parameter size	Obtained from the parameter definition or, if not specified there, a default value is used for some data types; else, you must specify it with the length property (and precision property, if necessary). See page 2-13 .	xfParameter	length=xxx precision=xxx
Parameter direction	Obtained from parameter modifiers (IN, OUT, INOUT) in the source code. If not supplied, defaults to IN. See "Parameter definitions" in the <i>"Defining Data"</i> chapter of the <i>Synergy Language Reference Manual</i> .	N/A	N/A
Parameter required/optional	Obtained from parameter modifiers (OPT, REQ) in the source code. If not supplied, defaults to required. (For .NET and Java clients, any parameter marked optional in the SMC is changed to required when the component is built.)	N/A	N/A

xfMethod Attribute

```
{xfMethod(property=value, property=value, ...)}
```

The `xfMethod` attribute describes a subroutine or function (Synergy OO methods are not supported). It indicates that the subroutine or function following the attribute is intended for use with `xfServerPlus` and should be included in the generated XML when `dbl2xml` is run. `xfMethod` must be used before *each* subroutine or function that you want included in the SMC. A routine may have more than one `xfMethod` attribute to indicate that it needs to be included in more than one interface. The properties that can be used with `xfMethod` are listed below. The interface and `elb` properties are required. Depending on the routine, you may need to specify other properties as well.

interface="name"

(required) The interface in which you want this method to be included. The quotation marks are required. Valid values for interface name are alphanumeric characters and the underscore character (`_`); it must begin with an alpha character. The interface name is case-sensitive within the SMC, but see the note on [page 2-25](#).

You may have only one instance of the interface property within an `xfMethod` attribute, so if a routine is to be included in more than one interface, you must create a separate instance of the `xfMethod` attribute for that routine. See [example B on page 2-16](#).

Methods are grouped into interfaces for inclusion in a Synergy component. The interface name will be used to select interfaces to include in the component, and users will see it as the class name when they use your JAR file, type library, or assembly. Although the interface is not used by `xfNetLink` Synergy, it is required for the XML that `dbl2xml` generates. Since interface name has no meaning for `xfNetLink` Synergy, you can use the same interface name for all methods if desired.



To reduce the amount of typing you have to do, you may want to use `.DEFINEs` to specify values for properties that occur numerous times in your code, such as interface name and `elb` name. For example,

```
.define myinterface interface="Login"  
.define myelb elb="EXE:utils"  
  
{xfMethod(myinterface, myelb)}
```

elb="path"

(required) The ELB or shared image in which the Synergy subroutine or function is implemented. You may use a complete path or a logical. The file extension (.elb or .exe) is not required. Maximum length is 255 characters.



If you use a logical in the elb property, you must define the logical in the **xfpl.ini** file (**SERVER_INIT.COM** on OpenVMS) so that *xfServerPlus* knows how to resolve it. See ["Defining Logicals" on page 1-4](#).

name="methodName"

Overrides the default method name, which is the same as the Synergy routine name. Valid values for method name are alphanumeric characters and the underscore character (_). The method name must begin with an alpha character. The maximum length is 50 characters. The name must be unique for the interface. (This comparison is case insensitive.) The name value is also used as the method ID, unless the id property is specified. (See below.) If the method name is longer than the maximum size of the method ID (31 characters), it is truncated to create the ID. Note that truncation could result in non-unique IDs, which are not permitted. For more information about method name and routine name, see ["Understanding Routine Name, Method Name, and Method ID" on page 2-2](#).



Although the \$ character is valid in routine names in Synergy Language, it is not valid for method names (or method IDs) in the SMC. If you use \$ in your routine names, be sure to use the name property to specify a valid method name.

id="methodID"

Overrides the default method ID, which is either the Synergy routine name or, if the name property is used, the method name. Valid values for method ID are alphanumeric characters and the underscore character (_). The method ID must begin with an alpha character. The maximum length is 31 characters. The method ID must be unique for the SMC.

In most circumstances, you will not need to specify the id property. But there may be cases where defaulting the method ID from the routine name or method name results in a non-unique method ID; in these cases, you need to specify the id property. See [example B on page 2-16](#) for an instance in which you must use the id property. Because the method ID must be compared with any method IDs that

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

may already be in the SMC, the check for uniqueness cannot take place until the XML file is imported into the SMC. For more information on the method ID and how it is used by *xyServerPlus*, see [“Understanding Routine Name, Method Name, and Method ID” on page 2-2](#).

length=##

precision=##

The size of the function return value. For most data types, the size is obtained from the function definition, and if the size cannot be so obtained, a default value is used. (See table below.) You can override this default by specifying the size in the function definition or with the length and (for implied-decimal) the precision properties. For alpha data types, you *must* use the length property if the size is not specified in the function definition, as there is no default value. Supported data types that are not included in the table below either have a size of 0 in the XML or have a default size that cannot be overridden.

Data type	Default if not specified in code
Alpha (a)	N/A - Must be specified in code or with the length property.
Decimal (d)	18
Implied-decimal (d.) Decimal (decimal)	28.10
Integer (i, int, or integer)	4
^VAL	4

cType=xfType.ret_type

Specifies a non-default data type for the return value to be coerced to on the client side. This feature is supported for .NET clients only. Decimal, implied-decimal, and integer data types can be coerced. See the table below for the valid *ret_type*

values for each data type. See [example D on page 2-17](#). (This example shows type coercion for a parameter, but the principal is the same for the return value.) See [Appendix B](#) for more information on data type mapping and coercion.



Data types `byte`, `sbyte`, `short`, `int`, `long`, and `Boolean` are built-in data types in Synergy Language that map to integers. Consequently, you can use these data types directly in the function definition, rather than specifying an integer and then using the `cType` property to specify a non-default coerced type. See [“Data types”](#) in the *“Defining Data”* chapter of the *Synergy Language Reference Manual* for more information on these types.

Return value data type	Valid coerced types
Decimal (d)	byte short int long sbyte ushort uint ulong Boolean DateTime
Implied-decimal (d.)	decimal double float
Integer (i)	byte short int long sbyte ushort uint ulong Boolean

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

format=xfFormat.*format*

The format for a DateTime return type. The format property is required when the cType property is DateTime. The valid values for *format* are shown below (not case sensitive).

- ▶ YYYYMMDD
- ▶ YYMMDD
- ▶ YYYYJJJ
- ▶ YYJJJ
- ▶ HHMMSS
- ▶ HHMM
- ▶ YYYYMMDDHHMISS

nullable=true

Indicates that a nullable DateTime return type is desired on the client side. The nullable property is valid only when the cType property is DateTime. “False” is a valid value and is the same as not setting the property.

encrypt=true

Indicates that encryption is desired for this method. This feature is supported for .NET and Synergy clients only. If you are using slave encryption and want the parameter and return value data for this method to be encrypted, set the encrypt property to true. “False” is a valid value and is the same as not setting the property. See [“Using Encryption” on page 3-23](#) for more information.

xfParameter Attribute

```
{xfParameter(property=value, property=value, ...)}
```

The xfParameter attribute describes the parameters in a routine. xfParameter is not required; use it when the necessary metadata for the parameter cannot be determined from the code. Only one instance of the attribute is permitted per parameter. The properties that can be used with xfParameter are listed below.

name="paramName"

Overrides the default parameter name, which is the name of the declared Synergy parameter. Valid values for parameter name are alphanumeric characters and the underscore character (_). The name must begin with an alpha character. The maximum length is 50 characters. The name must be unique for the method. (This comparison is case insensitive.)



Although the \$ character is valid in parameter names in Synergy Language, it is not valid in parameter names in the SMC. If you use \$ in your parameter names, be sure to use the name property to specify a valid name for the SMC.

type=SynType.data_type

Specifies the data type. The valid values for *data_type* are shown below:

- ▶ **handle**. Indicates non-array data of variable length or larger than 64k. See [example F on page 2-18](#).
- ▶ **binaryhandle**. Indicates binary data such as a JPEG file or Synergy RFA. See [example G on page 2-18](#).

The Synergy parameter must be a memory handle (**i4**; do not use **int**) when using this property. For more information on using a memory handle to pass large, variable length, or binary data see the following:

- ▶ [“Passing a Single Parameter as a Memory Handle” on page 1-12](#)
- ▶ [“Passing Binary Data” on page 1-18](#)

length=##

precision=##

The size of the parameter. If the size is included in the parameter definition, it is used. If the size is not included in the definition, a default value is used for some data types. (See table below.) You can override this default by specifying the size in the definition or with the length property (and, for implied-decimal data, the precision property). For alpha data types, you *must* use the length property if the size is not specified in the parameter definition, as there is no default value. This includes System.Collections.ArrayList parameters in which the elements are alphas.

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

Supported data types that are not included in the table below either have a size of 0 in the XML or have a default size that cannot be overridden.

Data type	Default size
Alpha (a)	N/A - Must be specified in parameter definition or with the length property.
Decimal (d)	18
Implied-decimal (d.) Decimal (decimal)	28.10
Integer (i, int, or integer)	4
Numeric (n)	18
Implied numeric (n.)	28.10



Not all parameter data types are supported on all clients. See the table on [page 2-32](#).

cType=xfType.data_type

Specifies a non-default data type for the parameter to be coerced to on the client side. Decimal, implied-decimal, and integer data types can be coerced. This feature is supported for .NET clients only. See the table on [page 2-11](#) for the valid cType values for each data type. See [example D on page 2-17](#). See [Appendix B](#) for more information on data type mapping and coercion.



Data types byte, sbyte, short, int, long, and Boolean are built-in data types in Synergy Language that map to integers. Consequently, you can use these data types directly in the parameter definition, rather than specifying an integer and then using the cType property to specify a non-default coerced type. See [“Data types”](#) in the *“Defining Data”* chapter of the *Synergy Language Reference Manual* for more information on these types.

format=xfFormat.format

Indicates the format for a DateTime parameter. The format property is required when the cType property is DateTime. See [page 2-12](#) for the list of valid values for *format*.

nullable=true

Indicates that a nullable DateTime parameter is desired on the client side. The nullable property is valid only when the cType property is DateTime. “False” is a valid value and is the same as not setting the property. See [example E on page 2-18](#).

collectionType=xfCollectType.data_type

Indicates the data type of the elements when the parameter is a System.Collections.ArrayList (supported on .NET clients only) or a structure collection (supported on Java and .NET clients only). For structure collections, the collectionType property is always “structure” and is used in conjunction with the structure property (see below). For an ArrayList of structures, you must also use the structure property to specify the structure name.

Valid values for *data_type* are as follows:

- ▶ alpha
- ▶ decimal (for decimal or implied-decimal)
- ▶ integer
- ▶ string (for System.String)
- ▶ structure

See [example H on page 2-18](#) for a structure collection, [example C on page 2-17](#) for an ArrayList of structures, and [example I on page 2-19](#) for an ArrayList of alphas.

structure="structureName"

Specifies the structure name when the parameter is a structure collection or ArrayList of structures. Used only in conjunction with the collectionType property. See [example H on page 2-18](#).



If you rename structures in a .INCLUDE like this

```
.include "STRUCT_1" REPOSITORY, structure="STRUCT_2" ,end
```

use the new name (STRUCT_2 in our example) in the property, not the original name. The **dbl2xml** utility will map the name in the property to the original name and write the latter to the XML. See [example C on page 2-17](#).

dataTable=true

Indicates that the `System.Collections.ArrayList` or structure collection parameter should be created as a `DataTable` on the client. This feature is supported on .NET clients only. “False” is a valid value and is the same as not setting the property.

Used only in conjunction with the `collectionType` property. See [“Using DataTables” on page 17-18](#) for more information on `DataTables`.

Attribute Examples

- A. This is a basic example that shows the `xfMethod` attribute for a function. The `xfMethod` attribute includes the interface name and ELB path (which uses a logical); no other properties are required. The method and method ID will default to the function name. No `xfParameter` attribute is required because all the necessary parameter information (name, direction, data type, size) is included in the definition.

```
{xfMethod(interface="ConsultApp", elb="EXE:Consult")}  
function ReturnError ,string  
    req in userToken          ,a22  
endparams
```

- B. This example shows the `xfMethod` and `xfParameter` attributes for a function that is included in two interfaces. The method name (Login) will be the same in both interfaces. The method ID for the first `xfMethod` attribute will default from the method name, but we must supply an ID for the second attribute, as the ID must be unique.

For the three parameters, the `xfParameter` attribute includes the name property, since the declared parameter name includes the character \$, which is invalid in the SMC. The third parameter is a structure; the associated repository structure is `.INCLUDED`. When using `dbl2xml`, structures passed as ordinary parameters and as arrays must be defined as structfields in your Synergy code. (See [“Structure”](#) in the “Defining Data” chapter of the *Synergy Language Reference Manual* for information on structfields.)



When an attribute follows a `.INCLUDE`, as in the example below, the `END` qualifier is required. (Normally, when you `.INCLUDE` a global structure, the `END` qualifier is not required, though it is recommended, and the compiler will issue a warning if it's not specified.)

```
.include "USER" REPOSITORY, structure, end

{xfMethod(interface="Customer", name="Login",
elb="EXE:Consult")

{xfMethod(interface="Vendor", name="Login", id="loginV",
elb="EXE:Consult")

function alogin ,^val

{xfParameter(name="userID")
    req in a$id          ,a10      ;User id

{xfParameter(name="userPword")
    req in a$password    ,a8        ;User password

{xfParameter(name="userData")
    req out a$user       ,USER      ;User info record

endparams
```

- C. This example shows a function with two parameters. The function returns a Boolean data type (see the tip on [page 2-11](#)). For the first parameter, we don't need an attribute because all the necessary information is in the code. The second parameter is an ArrayList in the Synergy code; on the client, we want to create an ArrayList of structures as a DataTable. To accomplish this, we specify the collectionType property (which specifies the type of elements in the ArrayList), the structure name, and the dataTable property. Note that we use the redefined structure name (country), not the original name (cntry) in the property.

```
.include "CNTRY" REPOSITORY, structure="country", end

{xfMethod(interface="ConsultApp", name="getCountryTable",
elb="EXE:Consult")

function get_country_table ,boolean

    req in  userToken      ,a22

    {xfParameter(collectionType=xfCollectType.structure,
structure="country", dataTable=true)}
    req out countryTable   ,@System.Collections.ArrayList

endparams
```

- D. This parameter example shows a parameter defined as a d8, which is coerced to a DateTime data type, using the cType and format properties.

```
{xfParameter(cType=xfType.DateTime, format=xfFormat.YYYYMMDD)}

req out updateDate ,d8
```

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

- E. This example is the same as above, but the DateTime data type will be created as a nullable DateTime on the client:

```
{xfParameter(cType=xfType.DateTime, format=xfFormat.YYYYMMDD,
nullable=true)}

req out updateDate ,d8
```

- F. This parameter example shows a parameter that will be used to pass large or variable size data. The parameter is a memory handle (i4) in the Synergy code, and we set the type property to “handle”. (See [“Passing a Single Parameter as a Memory Handle” on page 1-12](#) for more information on using this feature.)

```
{xfParameter(name="largeParam", type=SynType.handle)}

req inout memHandle ,i4      ;Mem handle for large param
```

- G. This parameter example shows a parameter that will be used to pass binary data. The parameter is a memory handle (i4) in the Synergy code, and we set the type property to “binaryhandle”. This will result in a byte array on the .NET client. (See [“Passing Binary Data” on page 1-18](#) for more information on using this feature.)

```
{xfParameter(name="fileData", type=SynType.binaryhandle)}

req out memHandle ,i4      ;Mem handle for binary data
```

- H. This parameter example shows a parameter that will be used to pass a structure collection, which will be created as a DataTable on the client. In the Synergy code, the parameter is defined as a memory handle (see [page 1-14](#).) The xfParameter attribute includes the structure property (for the structure name), the collectionType property to indicate the data type (which is structure in this case), and the dataTable property.

```
.include "USER" REPOSITORY, structure, end
.
. ;routine definition and xfMethod attribute go here
.

{xfParameter(name="CustList", structure="User",
collectionType=xfCollectType.structure, dataTable=true)}

req out memHandle ,i4      ;Mem handle for structure collection
```


- I. This parameter example shows an ArrayList of alphas. The `collectionType` property specifies that the data type of the elements in the ArrayList is alpha. Because there is no default size for alphas, we include the `length` property.

```
{xfParameter(collectionType=xfCollectType.alpha, length=30)}
```

```
req inout cityList ,@System.Collections.ArrayList
```
- J. This parameter example shows an array of structures and an alpha array. The arrays must be real arrays, not pseudo arrays or dynamic arrays. Include the size of an array element in the parameter definition or with the `length` property. The structure must be defined as a structfield. Because all the information is included in the parameter definition, the `xfParameter` attribute is not required.

```
.include "USER" REPOSITORY, structure, end
.
. ;routine definition and xfMethod attribute go here
.
req inout myStructArray ,[*]user ;Array of structures
req inout myAlphaArray ,[*]a10 ;Alpha array
```

Documentation Comments

When attributing your code, you can include documentation comments, which will be processed by **dbl2xml**, included in the XML file, and then imported into the SMC. When you generate classes for *xy*NetLink Java or *xy*NetLink .NET, the comments are included in the generated code. You can then use the comments to generate Javadoc or API documentation. Documentation comments are not supported for *xy*NetLink Synergy and *xy*NetLink COM. (Their presence in the SMC does not represent an error condition; they are just ignored.)

For details on using comments to generate documentation, see the following:

- Java, see [“Generating Javadoc” on page 9-16](#)
- .NET, see [“Generating API Documentation” on page 16-27](#)

You can add comments for methods, return values, and parameters. Each type of comment is distinguished by a particular tag, as explained in [“Comment tags”](#) below. Note the following:

- Start each comment line, including lines that contain only comment tags, with three semi-colons:

```
;;; This is a comment
```

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

- ▶ Each type of comment is limited to 6 lines of 50 characters each (not counting the three semi-colons).
- ▶ The comment text may include numbers, letters, and special characters.

Comment placement

Comments apply to the routine that they precede. For readability, we recommend that you put them together in a block either immediately before or immediately after the associated `xfMethod` attribute.

Comment tags

<summary> </summary>

Use the `<summary>` tag for comments that describe the function or subroutine. Only one tag is permitted per routine.

<returns> </returns>

Use the `<returns>` tag for return value comments. Valid only for functions. Only one tag is permitted per function.

<param name="*paramName*"> </param>

Use the `<param>` tag for parameter comments. Use a separate tag for each parameter and use the name property to specify which parameter the comment pertains to. The name in the comment tag should match the value of the name property in the `xfParameter` attribute, if it is used; else, it should match the name of the parameter in the source code.

Examples

The example below includes comments for the method (summary), return value, and each of the three parameters. Comments longer than 50 characters are broken into two lines. Note that for the third parameter, the name in the `<param>` tag matches the name specified with the name property of the `xfParameter` attribute.

```
{xfMethod(interface="ConsultApp", elb="EXE:Consult")}  
;;;<summary>  
;;;Logs user into application and verifies ID and  
;;; password  
;;;</summary>  
  
;;;<returns>  
;;;Indicates success or failure  
;;;</returns>
```

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

```
;;;<param name="a_id">
;;;User ID
;;;</param>

;;;<param name="a_password">
;;;User password
;;;</param>

;;;<param name="User">
;;;User record containing user first name, last name,
;;; and maximum billing rate.
;;;</param>

.function alogin, ^val
req in a_id      ,a10
req in a_password ,a8
{xfParameter(name="User")}
req out a_user   ,user   ;USER structure
endparams
```

Using the MDU to Define Synergy Methods

The Method Definition Utility (MDU) can be used to add, change, and delete data in the Synergy Method Catalog. The utility displays a list of all methods in the SMC and has two main data input screens: one for entering information about the method and one for entering information about each parameter. The MDU includes a search function for both methods and parameters, as well as options to import and export data (see [“Importing and Exporting Methods” on page 2-39](#)) and to verify repository structure sizes (see [“Verifying Repository Structure Sizes and Enumerations” on page 2-42](#)).

On Windows, you can start the MDU from the Windows Start menu, from Workbench, or by double-clicking on `mdu.dbr` from Windows Explorer. On all platforms, you can start the MDU using the command line syntax; see [“The Method Definition Utility” on page 2-50](#).

Creating New Methods



Data about your Synergy routines *must* be entered correctly in the SMC! In addition, the SMC *must* remain in sync with your Synergy code. If you change your code (e.g., change the data type of a function return value, add a new parameter, or alter a structure size in the repository), don't forget to update the SMC. (You can quickly update structure size changes with the Verify Catalog utility; see [page 2-42](#).)

1. Start the Method Definition Utility. See [page 2-50](#) for the command line syntax.

When the MDU opens, the methods list (see [figure 2-1](#)) displays the methods already in the catalog. The SMC is pre-loaded with several methods: XFPL_LOG, XFPL_REGCLEANUP, and a number of methods needed to run the *xfNetLink* and *xfServerPlus* test programs. (The methods for the test programs are in the interface *xfTest*.)

By default, the methods list displays the method name. You can toggle between displaying the method name and the method ID, by pressing CTRL+V or selecting Functions > Toggle View.

Data displayed on the methods list can be sorted by interface name (the default), method name (or method ID, depending on the toggle view setting), routine name, or ELB/shared image name. To change the sort order, click on the column headings (Windows only) or use the options on the Sort menu.

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

Interface	Method name	Routine	ELB
	XFPL_LOG	XFPL_LOG	DBLDIR:xfp1_api
	XFPL_REGCLEANUP	XFPL_REGCLEANUP	DBLDIR:xfp1_api
AppLogin	login	alogin	EXE:consult
AppLogin	logout	alogout	EXE:consult
Consultant	checkRecord	check_record	EXE:consult
Consultant	postCharge	post_charge	EXE:consult
xfTest	function_eight	function_eight	DBLDIR:xfp1_tst
xfTest	function_four	function_four	DBLDIR:xfp1_tst
xfTest	function_nine	function_nine	DBLDIR:xfp1_tst
xfTest	function_one	function_one	DBLDIR:xfp1_tst
xfTest	function_ten	function_ten	DBLDIR:xfp1_tst
xfTest	function_two	function_two	DBLDIR:xfp1_tst
xfTest	get_verno_tst	get_verno_tst	DBLDIR:xfp1_tst

Sorted by interface

Figure 2-1. Displaying the methods list.

2. To add a new method do one of the following:

- ▶ Click the Add button.
- ▶ Press the INSERT key.
- ▶ Select Functions > Add Method.

The Method Definition window will display (see [figure 2-2](#)).

3. Specify the following information in the Method Definition window.

Method name. Enter a name for the method. The method name must be unique for the interface in which it is included. (This comparison is case insensitive.) Valid values for this field are alphanumeric characters and the underscore character (_). The method name must begin with an alpha character.

When you exit the Method name field, the MDU copies the value you entered to the Method ID field. If the method name is longer than 31 characters (which is the maximum length for method ID), it will be truncated. If the MDU cannot create a valid and unique method ID from the method name, it will prompt you to supply a method ID. See [“Specifying a Method ID”](#) on page 2-28 for instructions on specifying or changing the method ID.

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

The screenshot shows a window titled "Method Definition" with a close button in the top right corner. The window is divided into two main sections: "xFNetLink method information" and "Synergy routine information".

xFNetLink method information:

- Method name: login
- Interface name: AppLogin
- Method ID: login
- Method desc: Verifies ID and password entered by user (with a dropdown arrow)
- ☐ LINQ stored procedure

Synergy routine information:

- Routine name: alogin
- ELB name: EXE:consult
- Return type: Decimal (dropdown menu)
- Length: 1
- Precision: (empty field)
- Coerced type: Boolean (dropdown menu)
- DateTime format: (empty dropdown menu)
- Return desc: Returns success or failure (with a dropdown arrow)

At the bottom, there is a checkbox for "Enable encryption" and two labels: "Maximum number of parameters" with the value "3" and "Number of required parameters" with the value "2". At the very bottom are four buttons: "OK", "Cancel", "Method ID", and "Parameter".

Figure 2-2. Completing the Method Definition window.

See [“Understanding Routine Name, Method Name, and Method ID”](#) on page 2-2 for details on the role of the method name and how it differs from method ID.

Interface name. (optional) If this method is going to be included in a Java JAR file, a COM type library, or a .NET assembly, enter the name of the interface that this method is part of. Valid values for this field are alphanumeric characters and the underscore character (_). The interface name must begin with an alpha character. This field is case sensitive.

Methods are grouped into interfaces for inclusion in a Synergy component. The interface name will be used to select interfaces to include in the component, and users will see it as the class name when they use your JAR file, type library, or assembly.

The interface name is also used to select methods when you generate test skeletons. (See [“Testing Your Synergy Code”](#) on page 1-20.) Although this field is optional for xFNetLink Synergy users, if you want to generate test skeletons, you must complete it.



Although the Interface name field is case sensitive, we do not recommend creating interface names that differ only in case. If you attempt to generate a component that contains two or more interfaces that differ only in case, the **genjava**, **gentlb**, or **gens** utility will append a number (starting with 1 and incrementing) to the end of each of the additional interfaces to make the class names unique (e.g., MYINTERFACE, MyIntFace1, myintface2, etc.). You will also see numbers appended to class names when there is a structure name that is the same as an interface name. Because structure classes are processed first, it is the procedural class name that will be altered.

If you are using *xfNetLink .NET*, the interface name *must* be different than the method name. Matching interface and method names will result in a “member names cannot be the same as their enclosing type” error when the classes are compiled.

Method desc. (optional) Enter a description of the method. If you are using a Java or .NET client, the information in this field will be included in the generated code as a documentation comment and, subsequently, in your generated Javadoc or API documentation, should you choose to create it. (See “[Generating Javadoc](#)” on page 9-16 or “[Generating API Documentation](#)” on page 16-27 for more information on generating documentation.) If you are using a Synergy or COM client, this field is ignored.

The Method desc field will hold six lines of 50 characters each. Only the first line can be edited on the Method Definition window. To access all six lines, press F9 or select Functions > Edit Description while your cursor is in this field. (On Windows, you can also click the drilldown button.) The Description dialog box will display. Text in this dialog box word-wraps automatically. Pressing Return will move the cursor to the next line, resulting in a line break that will be preserved in the generated code. Special characters (e.g., < and >) will be handled by **genxml**, **genjava**, and **gens**, so you should enter these characters as you want them to appear in the Javadoc or API documentation. When you have completed the description text, select OK or press F3 in the Description dialog box to save the text and return to the Method Definition window.

LINQ stored procedure. This feature is not yet implemented.

Routine name. Enter the name of the Synergy subroutine or function. Valid values are alphanumeric characters, the underscore character (_), and the dollar sign (\$). The routine name must begin with an alpha character

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

ELB name. Enter the name of the ELB or shared image in which the Synergy routine is stored. Include the full path or logical; don't include the **.elb** or **.exe** filename extension.

On UNIX and OpenVMS, if the ELB or shared image name is longer than 51 characters, press F10 or select Functions > Edit ELB Name while your cursor is in this field to display the ELB Name dialog box. Enter the complete ELB or shared image name and press F3 or select Main > Exit to save changes and return to the Method Definition window.



If you use logicals in the ELB name field, you must define them in the **xfpl.ini** file (**SERVER_INIT.COM** on OpenVMS) so that *xfServerPlus* knows how to resolve them. See [“Defining Logicals” on page 1-4](#).

Return type. If you are defining a function, select the Synergy Language data type of the returned data. For subroutines, select “No return value”. System.String and Enumeration are supported only on .NET clients; all other return types are supported on all clients.

If you select Enumeration, a list of enumerations defined in the current repository will display. Select an enumeration from the list by highlighting it and pressing ENTER (on Windows, you can also double-click).



While the list of enumerations is displayed, you can select Functions > Find to locate an enumeration in the list.

The MDU will display the enumeration name in a read-only field below the return type.



To check the location of the current repository, select Main > Repository Location. If the fields are blank, the MDU was unable to locate a repository on start-up. To change the repository, you must restart the MDU. See the MDU syntax on [page 2-50](#).

Length. If you specified a return type, enter the size of the return value. Note the following:

- ▶ For decimal return values coerced to DateTime or nullable DateTime data types, the length is determined by the selected DateTime format.
- ▶ For System.String data types, this field is set to zero.
- ▶ For Enumeration data types, this field is set to 4.

Precision. If you specified that the return type is implied-decimal, enter the precision. For example, if the return value is a **d6.2**, enter 6 in the Length field and 2 in the Precision field.

Coerced type. If the return type is decimal, implied-decimal, or integer, you can optionally select a non-default data type for the return value to be coerced to on the client side. Supported on .NET clients only. Select “Default” to use default type mapping. See [Appendix B](#) for more information on data type mapping and coercion.

DateTime format. If the coerced type is DateTime or nullable DateTime, select the desired format. Supported on .NET clients only.

Return desc. (optional) Enter a description of the return value. If you are using a Java or .NET client, the information in this field will be included in the generated code as a documentation comment and, subsequently, in your generated Javadoc or API documentation, should you choose to create it. If you are using a Synergy or COM client, this field is ignored. See the description of the Method desc field on [page 2-25](#) for details on creating a multi-line description.

Enable encryption. Select this checkbox if you are using slave encryption and want the parameter and return value data for this method to be encrypted. See [“Using Encryption” on page 3-23](#) for more information.

4. Select OK or press F3 to save your work.
5. If this is a new method, you will be prompted to define parameters. If there are no parameters for this method, select No at the prompt. If there are parameters, select Yes and see [“Defining Parameters” on page 2-28](#) for instructions.

Specifying a Method ID

When you enter a method name on the Method Definition window, the MDU copies that value to the Method ID field, which is a display-only field on the Method Definition window. You can edit or change the method ID by accessing the Method ID dialog box.



The method ID must be unique. If the MDU cannot create a valid and unique method ID by copying the value in the Method name field, it will prompt you to supply your own method ID and display the Method ID dialog box.

1. When the Method Definition window is displayed, select the Method ID button, or press F8, or select Functions > Method ID to display the Method ID dialog box.
2. Enter a method ID. This field is case sensitive. The method ID must be unique for the method catalog. Valid values for this field are alphanumeric characters and the underscore character (_). The method ID must begin with an alpha character.
3. Select OK or press F3 in the Method ID dialog box to save changes and return to the Method Definition window.

Defining Parameters

Follow these instructions to define parameters for a new method or add parameters to an existing method. A method can have a maximum of 253 parameters. You should define parameters in the order in which they are passed; the MDU assigns a sequence number to each parameter as you define it. See [“To resequence parameters” on page 2-35](#) for instructions on changing the sequence.

The first time you define parameters for a method, the Parameter Definition window displays in creation mode, as shown in [figure 2-3](#). When this window is in creation mode, each time you select the Next button (or press CTRL+N or select Functions > Next Parameter), the data you entered will be saved and the window refreshed so that you can enter another parameter. When you have entered all parameters for the method, select Done.

If you are adding parameters to a method that already has parameters, the Parameters list window displays the currently defined parameters. Highlight the line *above* where you want the new parameter to be added, and then click the Add button, press the INSERT key, or select Functions > Add to display the Parameter

Definition window in modify mode. When the window is in modify mode, selecting OK will save the parameter data and return you to the Parameters list window.



It is crucial that you define parameters correctly in the SMC! When making a call, *xfServerPlus* checks what you send in the call against what you defined in the SMC. If there are discrepancies, *xfServerPlus* will signal a non-fatal error. *xfServerPlus* cannot check what you send against what the Synergy routine requires, nor can it check what is defined in the SMC against the Synergy routine. If you define parameters incorrectly in the SMC and send data as it is defined, the error will not be detected until the routine is called on the server, and the result may be a fatal error.

► To define parameters

1. Complete the fields in the Parameter Definition window.

Figure 2-3. Completing the Parameter Definition window.

Parameter name. Enter an identifying name for the parameter.

Valid values for this field are alphanumeric characters and the underscore character (_). The parameter name must begin with an alpha character and must be unique for the method. (This comparison is case sensitive.) You may want to use the name of the Synergy argument as the parameter name.

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

If the parameter you are defining is a *structure*, press F7 or select Functions > Select Structure to display a list of structures in the current repository. Select a structure from the list by highlighting it and pressing ENTER (on Windows, you can also double-click).



While the list of structures is displayed, you can select Functions > Find to locate a structure in the list.

The MDU will enter the structure name as the parameter name and display the structure name in a read-only field below the data type. You can change the parameter name if desired. (If you enter a parameter name before displaying the list of structures, it will not be overwritten with the structure name.) The MDU also fills in the data type and the length with information from the repository.



You can also display the list of structures in the current repository by selecting "Structure" from the Data type selection list.

If the parameter you are defining is an *enumeration* (supported on .NET clients only), press F6 or select Functions > Select Enumeration to display a list of enumerations in the current repository. Select an enumeration from the list by highlighting it and pressing ENTER (on Windows, you can also double-click).



While the list of enumerations is displayed, you can select Functions > Find to locate an enumeration in the list.

The MDU will enter the enumeration name as the parameter name and display the enumeration name in a read-only field below the data type. You can change the parameter name if desired. (If you enter a parameter name before displaying the list of enumerations, it will not be overwritten with the enumeration name.) The MDU fills in the data type and the length with information from the repository.



You can also display the list of enumerations in the current repository by selecting "Enumeration" from the Data type selection list.



To check the location of the current repository, select Main > Repository Location. If the fields are blank, the MDU was unable to locate a repository on start-up. To change the repository, you must restart the MDU. See the MDU syntax on [page 2-50](#).

Description. (optional) Enter a description of the parameter. If you are using *xy*NetLink Java or *xy*NetLink .NET, the information in this field will be included in the generated code as a documentation comment and, subsequently, in your generated Javadoc or API documentation, should you choose to create it. (See “Generating Javadoc” on [page 9-16](#) or “Generating API Documentation” on [page 16-27](#) for more information on generating documentation.) If you are using *xy*NetLink Synergy or *xy*NetLink COM, this field is ignored.

This field will hold six lines of 50 characters each. Only the first line can be edited on the Parameter Definition window. To access the additional lines, press F9 or select Functions > Edit Description while your cursor is in this field. (On Windows, you can also click the drilldown button.) The Description dialog box will display. Text in this dialog box word-wraps automatically. Pressing Return will move the cursor to the next line, resulting in a line break that will be preserved in the generated code. Special characters (e.g., < and >) will be handled by **genxml**, **genjava**, and **gens** during code generation, so you should enter these characters as you want them to appear in the Javadoc or API documentation. When you have completed the description text, select OK or press F3 in the Description dialog box to save the text and return to the Parameter Definition window.

Data type. Select the Synergy Language data type of the parameter. If you selected a structure or enumeration while you were in the Parameter name field, the MDU sets this field to the correct data type. Not all parameter types are supported on all clients; see the table below.

For information on when to specify a parameter type of Handle, see “Passing a Single Parameter as a Memory Handle” on [page 1-12](#) and “Returning a Collection of Structures” on [page 1-14](#).

To pass binary data (such as a JPEG file or Synergy RFA), select “Binary (handle)”. See “Passing Binary Data” on [page 1-18](#) for more information.

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

Supported Parameter Data Types and Collection Types by Client				
Data/collection type in MDU	Synergy	Java	COM	.NET
Alpha	✓	✓	✓	✓
Decimal	✓	✓	✓	✓
Implied-decimal	✓	✓	✓	✓
Integer	✓	✓	✓	✓
Structure		✓ ^a	✓	✓
Handle	✓	✓	✓	✓
Binary (handle)		✓		✓
System.String				✓
Enumeration				✓
Array	✓	✓	✓	✓
ArrayList				✓
Structure collection		✓ ^a		✓
DataTable				✓

a. Requires class wrappers.

Length. Enter the size of the parameter. Note the following:

- ▶ For arrays and ArrayLists that are not made up of structure elements, enter the element length, not the total array size.
- ▶ For structures and enumerations, the Length field is completed automatically with information from the repository.
- ▶ For handle, binary (handle), and System.String parameters, this field is set to zero.
- ▶ For decimal parameters coerced to DateTime or nullable DateTime data types, the length is determined by the selected DateTime format.

Precision. If the data type is implied-decimal, enter the precision. For example, for a **d10.3** parameter, enter 10 in the Length field and 3 in the Precision field.

Coerced type. If the data type is decimal, implied-decimal, or integer, you can optionally select a non-default data type for the parameter to be coerced to on the client side. Supported on .NET clients only. Select “Default” to use the default type mapping. See [Appendix B](#) for more information on data type mapping and coercion.

DateTime format. If the coerced type is DateTime or nullable DateTime, select the desired format. Supported on .NET clients only.

Array. If the parameter is an array, select this field. The parameter must be defined as a real array in your Synergy code; if you use pseudo arrays, you should convert them to real arrays. Arrays are supported for all data types except handle, binary (handle), System.String, and enumeration.

x/NetLink Synergy supports multi-dimensional arrays (up to 9 dimensions); the number of dimensions and elements per dimension are calculated from the data passed. *x/NetLink* Java, *x/NetLink* COM, and *x/NetLink* .NET support only one-dimensional arrays. The total size of an array passed to the Synergy server may be greater than 64K, but each element in the array must be less than 64K. See [“Passing Arrays Larger Than 64K” on page 1-17](#) for more information.

ArrayList. If the parameter is a Synergy System.Collections.ArrayList class, select this field. Supported on .NET clients only. ArrayLists are supported for all data types except handle, binary (handle), and enumeration. When using an ArrayList, Data passed can be set to In or Out, but not In/Out. See [“Passing a System.Collections.ArrayList Parameter” on page 1-16](#) for more information.

Structure collection. If the parameter is a structure, indicate if you want to pass it as a structure collection. A structure collection is a memory handle on the *x/ServerPlus* side and an ArrayList on the client side. Supported on Java and .NET clients only. When you select this option, Data passed is set to Out, as structure collection parameters can be used only to pass data from Synergy to the client. See [“Returning a Collection of Structures” on page 1-14](#) for more information.

DataTable. If the parameter is a structure and you selected ArrayList or Structure collection, indicate if you want the parameter to be created as a DataTable on the client. Supported on .NET clients only. See [“Using Data Tables” on page 17-18](#) for more information.

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

Data passed. Indicate if the parameter is passed in, out, or both in and out.

- ▶ **In** indicates this parameter is used to pass data from the client to *xyServerPlus*. Select In when the parameter is sending input that is not changed by the Synergy routine. This is the default.
- ▶ **Out** indicates this parameter is used to pass data from *xyServerPlus* to the client. Select Out when the parameter is used to return output information created by the Synergy routine to the client.
- ▶ **In/Out** indicates that this parameter is used to pass data from the client to *xyServerPlus* and to return an updated version of that data to the client. Select In/Out when the parameter is sent as input, updated by the Synergy routine, and then returned to the client.

Pass by. Indicate if the parameter is passed by descriptor (i.e., normally), value (^VAL), or reference (^REF). The default is Descriptor. If the data type is handle or binary (handle), the MDU sets this field to Descriptor.

Required. If the parameter is required, select this field. The default is Required. If the parameter is optional, clear this field.

Optional parameters are not supported in *xyNetLink Java* and *xyNetLink .NET*. When **genjava** or **gencs** is run, any parameters marked as optional will be converted to required and a warning message will be generated. (This is because these languages do not support optional parameters.)

2. If the window is in creation mode and you need to add another parameter, select the Next button (or CTRL+N or Functions > Next Parameter). The information for the current parameter will be saved, and the window will be refreshed so that you can add another parameter.

If you are finished adding parameters, select the Done button (or F3 or Functions > Done). The Parameters list window will display, with the new parameter(s) added to the list.

3. If the window is in modify mode, select OK or press F3. The Parameters list window will display, with the new parameter added to the list.
4. Select Done or press F3 in the Parameters list window. The MDU calculates the total number of parameters and the number of required parameters and updates the count on the Method Definition window.
5. Select OK or press F3 in the Method Definition window.

► To resequence parameters

The numbers on the left side of the Parameters list window reflect the order in which the parameters are passed when this routine is called. New parameters are inserted below the parameter that is highlighted when you select Add. (The parameter sequence number also appears on the Parameter Definition window below the method name or method ID.)

To resequence parameters, highlight the parameter you want to move and do one of the following to move it to the correct location:

- Use the Move up and Move dn buttons in the Parameters list window.
- Press CTRL+U to move it up or CTRL+D to move it down.
- Use the Move Up and Move Down commands on the Functions menu.

Modifying Methods and Parameters

► To modify methods

Use this procedure to change any data on the Method Definition window or on the dialog boxes accessed from that window, such as the Method ID dialog box and the Description dialog boxes.



The Date Last Updated field in the upper right corner of the Method Definition window reflects the date that this record was last changed. It is updated automatically when you change and save the record.

1. In the methods list, highlight the method you want to change.
2. Select it by clicking the Modify button, pressing ENTER, or selecting Functions > Modify Method.
3. Modify the method data as needed. See [“Creating New Methods” on page 2-22](#) for details on completing the fields. For information on changing the method ID, see [“Specifying a Method ID” on page 2-28](#).
4. Select OK or press F3 to exit and save your changes.

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

► To modify parameters

1. In the methods list, highlight the method that the parameter is associated with and then click the Modify button, press ENTER, or select Functions > Modify Method.
2. On the Method Definition window, select the Parameter button, press F7, or select Functions > Parameters.
3. In the Parameters list window, highlight the parameter to change and then click the Modify button, press ENTER, or select Functions > Modify.
4. Modify the parameter data as needed. See [“Defining Parameters” on page 2-28](#) for details on completing the fields.
5. Select OK or press F3 in the Parameter Definition window.
6. Select Done or press F3 in the Parameters list window.
7. Select OK or press F3 in the Method Definition window.

Deleting Data from the SMC

► To delete an interface

This procedure deletes an interface and all methods associated with it.

1. In the methods list, highlight one of the methods that is in the interface you want to delete.
2. Select Functions > Delete Interface.
3. Select Yes at the prompt to confirm the deletion.

► To delete a method

This procedure deletes a method and all parameters associated with it.

1. In the methods list, highlight the method you want to delete.
2. Click the Delete button, press the DELETE key, or select Functions > Delete Method.
3. Select Yes at the prompt to confirm the deletion.

► To delete a parameter

1. In the methods list, highlight the method that the parameter is associated with and then click the Modify button, press ENTER, or select Functions > Modify Method.
2. On the Method Definition window, select the Parameter button, or press F7, or select Functions > Parameters.
3. In the Parameters list window, highlight the parameter you want to delete.
4. Click the Delete button, press the DELETE key, or select Functions > Delete.
5. Select Yes at the prompt to confirm the deletion.
6. Select Done or press F3 in the Parameters list window.
7. Select OK or press F3 in the Method Definition window.

Searching for Methods and Parameters

There are search functions associated with both the methods list and the Parameters list window.

1. In the methods list or the Parameters list window, click the Find button, press CTRL+F, or select Functions > Find.
2. For methods, select the search type. You can search for methods by method name (or method ID, depending on the toggle view setting), interface name, routine name, or ELB/shared image name. You can search for parameters only by parameter name.
3. Enter the search criteria. You can enter a partial name if desired.
4. Press ENTER. The methods list or Parameters list window redisplay, and the first entry that matches the search criteria is highlighted. (The search is case insensitive.)
5. To find the next entry matching the search criteria, press CTRL+N or select Functions > Find Next.

Setting the Catalog Location

You can change which SMC the MDU is currently updating without restarting the MDU. This function is available only when the methods list is the active window.



If you access the Catalog Location dialog box from other windows in the MDU, you can view the location of the current catalog but not change it.

1. From the methods list select Main > Catalog Location. The Catalog Location dialog box displays the location of the currently selected SMC.
2. Type in a new path or select Functions > Browse to select the directory. (On Windows, you can also click the drilldown button to browse for a directory.) To specify an SMC on a remote machine that is running *x/Server*, specify the node name in the path. For example, `c:\smcDir\@machineName`.
3. Select OK or press F3. If a catalog does not exist in the directory you specify, you will be prompted to create a new catalog in that location. Assuming that the file **DBLDIR:defaultsmc.xml** is present, the newly-created catalog will include the default methods.

Importing and Exporting Methods

Using the MDU's import/export feature, you can export entries from one SMC and then import them into a different SMC. The export feature writes all the methods in the SMC to an XML file. You can then use the import feature to replace the entire catalog, update and add selected interfaces, or add selected methods. You can perform an import/export from within the MDU (see below) or from the command line, without displaying the MDU user interface (see [page 2-41](#)).

You might use import/export to

- ▶ populate or update your SMC by importing interfaces in an XML file generated by the **dbl2xml** utility.
- ▶ import the default methods normally distributed with the SMC. You can import these methods from **defaultsmc.xml**. This file, located in DBLDIR, contains XFPL_REGCLEANUP, XFPL_LOG, and the methods required by the test programs.
- ▶ update your deployment SMC with data from your development SMC.
- ▶ make global updates, such as changing an ELB name, by exporting the catalog, searching and replacing in the XML file, and then re-importing it. When editing the contents of the XML file, take care not to alter the XML structure.
- ▶ create a new catalog that includes the methods in an XML file. This must be done from the command line. See [“To create new files by importing an XML file” on page 2-44](#) for details.

▶ To import methods

Before importing, you should verify that no one else is using the SMC files you are importing into.

1. Open the SMC that you want to import methods into. You can do this from Main > Catalog Location, or you can restart the MDU and specify the catalog location on the command line.
2. From the methods list, select Utilities > Import Methods.

Defining Your Synergy Methods

Importing and Exporting Methods

3. Complete the fields in the Import Details window:

Select mode. Select the type of import desired:

- ▶ **Replace entire SMC.** Select this option to replace the existing SMC with the data in the XML file. Note that this option *completely deletes the existing SMC* and then imports all the methods from the XML file.
- ▶ **Update interfaces.** Select this option to update individual existing interfaces in the current SMC, as well as add new interfaces.
- ▶ **Add new methods.** Select this option to add individual methods to the current SMC. You must use this option if you need to import only the XFPL_REGCLEANUP or XFPL_LOG method, as those methods are not in an interface.

XML file. Type the filename of the XML file that contains the data to import or select Functions > Browse to select it. (On Windows, you can click the drilldown button to select the file.) To import the default methods such as XFPL_REGCLEANUP, choose the **defaultsmc.xml** file located in your DBLDIR directory.

Log file. Enter a filename for the error log or accept the default name (**import_smc.log**). If no path is specified, the log will be created in the current working directory on OpenVMS and in the location specified with the TEMP environment variable on Windows and UNIX.

4. Select OK or press F3.

If you selected “Replace entire SMC”, the data in the XML file will be imported immediately. See [step 7](#).

If you selected “Update interfaces” or “Add new methods”, a list of items will display. On the Select Interfaces list, an X in the “In SMC” column indicates that the interface is present in the current SMC. If you select this interface for import, the existing interface in the SMC will be deleted and the one in the XML file will be imported.



If you selected “Add new methods”, you’ll see a list of either method names or method IDs, depending on the current toggle view setting. You can press CTRL+V or select Functions > Toggle View while this list is displayed to switch the display.

5. Press ENTER (on Windows, you can also double-click) to mark items on the list for import. An asterisk in the left column indicates an item is selected. To unselect an item, double-click or press ENTER again.
6. Once you've selected all the items, select OK or press F3 to complete the import.
7. The import function validates the information in the XML file before importing it. If errors are found, you'll see a message to that effect, a log file will be created, and the SMC will not be updated. Check the log file to see what the problems are, correct them in the XML file, and then re-import.

► To export methods

If your SMC includes references to repository structures or enumerations, the repository must be present to do an export. This is because the export function reads the structure and enumeration details from the repository.

Upon export, structure sizes in the SMC are checked against the corresponding structures in the repository and the utility returns an error if there are any size discrepancies or if there are missing structures or enumerations. You can use Verify Catalog utility to update the structure sizes in the SMC. See [“Verifying Repository Structure Sizes and Enumerations” on page 2-42](#) for details.

1. Verify that the current SMC is the one that you want to export methods from. You can select Main > Catalog Location to check (or change) the location of the current catalog.
2. From the methods list, select Utilities > Export Methods.
3. At the prompt, type a name for the XML file, and then select Save or press ENTER.

► To import and export methods from the command line

The **mdu -i**, **-u**, and **-e** options enable you to import and export SMC definitions from the command line without displaying the MDU user interface. You can use these options to include import/export functionality in a script or batch file.

When importing from the command line, you do not have as many options as you do when using the MDU user interface: you can either replace the entire SMC or update the SMC using the interfaces in the XML file. You cannot select individual interfaces or methods for import. See the MDU command line syntax on [page 2-50](#) for details.

Verifying Repository Structure Sizes and Enumerations

The Verify Catalog utility enables you to quickly update structure sizes in the Synergy Method Catalog after making changes in your repository. This utility compares the structures in the SMC against the structures in the repository and updates the structure sizes in the SMC if necessary. All changes are logged, as are any other problems that are encountered, such as structures missing from the repository.

The utility also verifies that enumerations selected as parameters or return values in the MDU are present in the current repository. If there are any discrepancies, they are recorded in the logfile.

You can run the utility from within the MDU or from the command line.

► To verify a catalog

1. Open the SMC that you want to verify. You can do this from Main > Catalog Location, or you can restart the MDU and specify the new catalog location on the command line.
2. From the methods list, select Utilities > Verify Catalog.
3. Provide a log file name or use the default, **smc_verify.log**. If no path is specified, the log will be created in the current working directory on OpenVMS and in the location specified with the TEMP environment variable on Windows and UNIX.
4. Select OK or press F3. If problems are found, a message to that effect is displayed. Check the log file to see which structure sizes were corrected and to check for problems that the utility could not correct.

► To verify a catalog from the command line

You can use the **mdu -v** option to run the Verify Catalog utility from the command line without displaying the MDU user interface. You can use the **-v** option to include verification functionality in a script or batch file. See the MDU command line syntax beginning on [page 2-50](#).

Defining Multiple Synergy Method Catalogs

By default, the Method Definition Utility and *xyServerPlus* read the SMC files from the DBLDIR directory. However, there may be times when you want to use an alternate SMC. For example, you could create additional SMCs to allow users to work with their own versions of data during testing or training. Or, you may want to have a development SMC and a production (deployment) SMC. Note that at runtime, each instance of *xyServerPlus* can reference only a single SMC.

To define an alternate SMC, or to use an SMC that is not in the default location, you need to

- ▶ create the SMC files in a different location.
- ▶ specify the location of the files when you run the MDU.
- ▶ set the XFPL_SMCPATH environment variable on your server machine so that *xyServerPlus* can find the SMC files.

Creating New SMC Files

The SMC consists of the data files **cdt.is?** and **cmpdt.is?**. Because you cannot change these filenames, additional SMCs must be created in separate directories. You can do this either by creating new files or by copying existing files.

▶ To create new files from within the MDU application

This procedure creates new SMC files in a new location. The new SMC will include the default methods (XFPL_REGCLEANUP, XFPL_LOG, and the methods for the test programs), which are imported from **DBLDIR:defaultsmc.xml**. If this file is not in DBLDIR, you'll see a message and the new SMC files will be empty.

1. Create a directory for the SMC files.
2. Start the MDU.
3. Go to Main > Catalog Location and type the new directory path in the Catalog Location field or select Functions > Browse to select a directory. (On Windows, you can also click the drilldown button to browse for a directory.) Select OK or press F3.
4. Select Yes at the prompt to create new files in the specified location.

Defining Your Synergy Methods

Defining Multiple Synergy Method Catalogs

► To create new files by importing an XML file

From the command line, you can create a new catalog and import method definitions from an XML file into it in one step using the **-i** option.

The newly created catalog will include all methods in the XML file. If the default methods such as XFPL_REGCLEANUP are not in the XML file, you can import them from **defaultsmc.xml** after the new catalog is created. (See [“Importing and Exporting Methods”](#) on page 2-39.)



You can also create a new catalog by running MDU with the **-u** option. However, because **-u** imports only methods in named interfaces, if there are methods in the XML file that are not assigned to an interface (e.g., XFPL_REGCLEANUP), they will not be imported.

These instructions assume that you already have an XML file, created either by exporting from the MDU (see [“To export methods”](#) on page 2-41) or by running **dbl2xml** (see [“Using Attributes to Define Synergy Methods”](#) on page 2-3).

1. Create a directory for the SMC files.
2. Run the MDU from command line with the **-i** and **-l** options. Specify the location of the directory you created for the SMC files, the path and filename of the XML file to import from, and the path and filename for the logfile. For example:

```
dbr DBLDIR:mdu c:\work\NewSMC -i c:\temp\catalog.xml  
-l c:\temp\myLog.txt
```

Note that this command will run without the MDU displaying on the screen. See [“Using the MDU to Define Synergy Methods”](#) on page 2-22 for details on the command line syntax.

► To copy existing files

When you copy files, all the data in the current SMC is copied to the new one. You can then delete some of the methods, if desired, or delete all the methods by clearing the copied files with **isload**. (See **isload** in the “Synergy DBMS” chapter of *Synergy Language Tools* for instructions.)

1. Create a directory for the SMC files.
2. Copy the existing files from DBLDIR to the new directory. The files are named **cdt.is?** and **cmpdt.is?**.
3. If desired, open the MDU in the new location and delete some of the methods.

Specifying Which SMC to Update

If you have more than one SMC, or an SMC that is not in the default location, you need to specify which SMC to update when running the MDU. There are two ways to do this.

- ▶ Specify the SMC location on the command line when you start the MDU. See [“Using the MDU to Define Synergy Methods” on page 2-22](#) for information on starting the MDU from the command line.
- ▶ Set XFPL_SMCPATH in the environment to point to the location of the SMC files. For instructions on setting environment variables on your operating system, see [“Setting Environment Variables and Initialization Settings”](#) in the “Environment Variables” chapter of *Environment Variables and System Options*. Note that if you have set XFPL_SMCPATH in the Windows registry, the `synrc` file (UNIX), or the `SERVER_INIT.COM` file (OpenVMS), the MDU will ignore it; only *xfServerPlus* reads XFPL_SMCPATH from those locations (see [“Setting the XFPL_SMCPATH Environment Variable for xfServerPlus” on page 2-46](#)).

If you don't specify the SMC location on the command line or with XFPL_SMCPATH, the MDU will load the files from DBLDIR. If necessary, you can then change the files being read by selecting Main > Catalog Location. See [“Setting the Catalog Location” on page 2-38](#).



To verify the location of the SMC you're updating, in the MDU select Main > Catalog Location.

Setting the XFPL_SMCPATH Environment Variable for *xfServerPlus*

If you're using more than one SMC, or an SMC that is not in the default location, you need to set the environment variable XFPL_SMCPATH on your Synergy server machine to point to the directory that contains the SMC files.

At runtime, each instance of *xfServerPlus* can reference only a single SMC; however, if you have multiple instances of *xfServerPlus* running on different ports, they can each reference a separate SMC if desired. This enables you to maintain, for example, SMC files for testing and development and separate SMC files for production.



This section explains only how to set XFPL_SMCPATH so that it can be read by *xfServerPlus*. To set XFPL_SMCPATH so that it can be read by the MDU, see [“Specifying Which SMC to Update” on page 2-45](#).

Setting XFPL_SMCPATH on Windows

On Windows, use the Synergy Configuration Program to set environment variables for *xfServerPlus*. You can set XFPL_SMCPATH for all instances of *xfServerPlus* or for a specific instance of *xfServerPlus*.

To specify an SMC on a remote machine that is running *xfServer*, specify the node name when you define the environment variable. For example, `c:\smcDir\@machineName`.

► To set XFPL_SMCPATH for all instances of *xfServerPlus*

This procedure sets XFPL_SMCPATH in the Windows registry under HKEY_LOCAL_MACHINE\SOFTWARE\Synergex\Synergy *xfServer*\Synrc.

1. Start the Synergy Configuration Program (from the Start menu, select Programs > SynergyDE > Utilities > Synergy Configuration Program) and go to the *xfServer/xfServerPlus* tab.
2. Select Default from the list of services, select the Modify Service button, and then select the Environment Settings button.
3. Select the Add button that is grouped with the “Settings for all services” list.
4. Type the variable name (XFPL_SMCPATH) and value in the Add Environment Setting dialog box, and select OK.

5. Select OK in the xfServer Information dialog box, and then select Apply in the Synergy Configuration Program.
6. You'll be prompted to stop and restart all services so that the new settings take effect. Select Yes.

► To set XFPL_SMCPATH for a specific instance of xfServerPlus

This procedure sets XFPL_SMCPATH in the Windows registry under HKEY_LOCAL_MACHINE\SOFTWARE\Synergex\Synergy xfServer\ *serviceName*\Synrc. This setting overrides the “all instances” setting for a specific instance of xfServerPlus.

1. Start the Synergy Configuration Program (from the Start menu, select Programs > SynergyDE > Utilities > Synergy Configuration Program) and go to the xfServer/xfServerPlus tab.
2. Select the service from the list, select the Modify Service button, and then select the Environment Settings button.
3. Select the Add button that is grouped with the “Settings for *service name*” list.
4. Type the variable name (XFPL_SMCPATH) and value in the Add Environment Setting dialog box, and select OK.
5. Select OK in the xfServerPlus Information dialog box, and then select Apply in the Synergy Configuration Program.
6. You'll be prompted to stop and restart the selected service so that the new settings take effect. Select Yes.



We *do not* recommend setting XFPL_SMCPATH in the **xfpl.ini** file because it overrides *both* settings shown above.

Setting XFPL_SMCPATH on UNIX

On UNIX, the *xfServerPlus* service reads settings from the environment and from the **synrc** file. There are **synrc** files at the system level (**/etc/synrc**) and at the user level (**/usr/username/.synrc**). At the system level, *xfServerPlus* supports both a generic **synrc** file and a port-specific **synrc** file. See below for details.



To include comments in the **synrc** file, precede the comment with a number sign (#).

- ▶ To specify the SMC location for all instances of *xfServerPlus*, do one of the following:
 - ▶ Set XFPL_SMCPATH as an environment variable. (Note that the MDU will also read XFPL_SMCPATH set in the environment.)
 - ▶ Set XFPL_SMCPATH in the generic **synrc** file (**/etc/synrc**). This setting overrides XFPL_SMCPATH set in the environment.
- ▶ To specify the SMC location for a specific instance of *xfServerPlus* by port number, create a file named **synrc.####**, where **####** is the port number that *xfServerPlus* is running on, and put it in the **/etc** directory. If you have multiple instances of *xfServerPlus* running on multiple ports, you can create a file for each instance and then set XFPL_SMCPATH differently in each file. This setting overrides XFPL_SMCPATH set either in the environment or in the generic **synrc** file.
- ▶ To specify the SMC location for *xfServerPlus* by user name, set XFPL_SMCPATH in the **.synrc** file for the user account that *xfServerPlus* sessions run under (**/usr/username/.synrc**). If you create several *xfServerPlus* accounts, each one could have a different **.synrc** file with different settings. Setting XFPL_SMCPATH in the user's **.synrc** file overrides XFPL_SMCPATH set in the environment, the generic **synrc** file, and the port-specific **synrc** file.



We *do not* recommend setting XFPL_SMCPATH in the **xfpl.ini** file because it overrides *all* settings shown above.

Setting XFPL_SMCPATH on OpenVMS

We recommend setting XFPL_SMCPATH in DBLDIR:SERVER_INIT.COM.

- ▶ To specify the SMC location for *all instances* of *xfServerPlus*, use

```
$ DEFINE/SYS XFPL_SMCPATH altdir
```

where *altdir* is the directory in which the SMC files are located.
- ▶ To specify the SMC location for a *specific instance* of *xfServerPlus*, use

```
$ DEFINE/TABLE=LNMRSDMSMGR_port /USER - XFPL_SMCPATH altdir
```

where *port* is the port number on which the specific *xfServerPlus* session is running, and *altdir* is the directory in which SMC files are located. This setting overrides the system-level setting.



We *do not* recommend setting XFPL_SMCPATH in the **xfpl.ini** file because it overrides *both* settings shown above. For additional information on defining **rsynd** logicals on OpenVMS, see “[Defining logical names](#)” in the “Configuring xfServer” chapter of the *Installation Configuration Guide*.

The Method Definition Utility

The Method Definition Utility (MDU) is used to add, change, and delete data in the Synergy Method Catalog. It also includes functions for importing and exporting data and for verifying repository structures and enumerations.

Syntax `dbr mdw [smc_path] [-r rps_path] | [-m rps_main -t rps_text]
[-e XMLfilename] | [-i XMLfilename -l logfile] | [-u XMLfilename
-l logfile] | [-v -l logfile] [-?]`

Arguments *smc_path*

(optional) Specify the directory where the SMC files are located. If not passed, the environment variable XFPL_SMCPATH is used (see [“Specifying Which SMC to Update” on page 2-45](#)). If that is not set, the SMC files in DBLDIR are used. If *smc_path* is specified, it must be the first argument.

If your SMC files are located on a remote machine that is running *xfServer*, include the node name in the path when starting the MDU. See the examples on [page 2-53](#).

If you specify an existing directory path, either on the command line or with XFPL_SMCPATH, and there are no SMC files in that location, you will be prompted to create them. If you specify no directory path and there are no files in DBLDIR, you will be prompted to create them in DBLDIR.

You can change the SMC directory from within the MDU. See [“Setting the Catalog Location” on page 2-38](#).

-r rps_path

(optional) Specify the directory where the repository files are located. The MDU will look for the files **rpsmain.ism** and **rpstext.ism** in this directory. If your repository files are located on a remote machine that is running *xfServer*, include the node name in *rps_path*.

If *rps_path* is not passed, the environment variables RPSMFIL and RPSTFIL are used to determine the two repository filenames. If they are not defined, the repository files in RPSDAT are used.

You cannot change the repository once the MDU is running; you must restart the MDU to specify a different repository. We recommend that you use only one repository per SMC.



The MDU will attempt to open the repository files on start-up. If you explicitly specify the repository path (**-r**) or main and text filenames (**-m** and **-t**) on the command line, and the repository cannot be found or either of the repository files cannot be opened, the MDU will generate an error and terminate. If you don't use the **-r** option or the **-m** and **-t** options, and the repository cannot be found or opened, the MDU will start anyway and the Repository Location dialog box will display blank.

-m *rps_main*

(optional) Specify the complete path and filename for the repository main file. Use with **-t**. If *rps_main* and *rps_text* are not passed, the environment variables RPSMFIL and RPSTFIL are used to determine the two repository filenames. If they are not defined, the repository files in RPSDAT are used. If your repository files are located on a remote machine that is running *xyServer*, include the node name in *rps_main*. See the examples on [page 2-53](#).

-t *rps_text*

(optional) Specify the complete path and filename for the repository text file. Use with **-m**. If *rps_main* and *rps_text* are not passed, the environment variables RPSMFIL and RPSTFIL are used to determine the two repository filenames. If they are not defined, the repository files in RPSDAT are used. If your repository files are located on a remote machine that is running *xyServer*, include the node name in *rps_text*. See the examples on [page 2-53](#).

-e *XMLfilename*

(optional) Export the SMC definitions to the specified file. The file will be created in the current working directory unless you specify the complete path. If the SMC includes structure parameters or enumeration parameters or return values, the repository must be present. This option exports the entire SMC and writes it to an XML file without displaying the MDU user interface. You can use the **-e** option to incorporate export functionality into a batch file. Cannot be used with **-i**, **-u**, or **-v**.

-i *XMLfilename*

(optional) Import all definitions in the specified XML file. Specify the complete path or use a logical if the file is not in the current working directory. This option *replaces the entire catalog* with the definitions in the specified XML file. (See **-u**, below, for an update option.) The MDU user interface does not display, so you can use the **-i** option to incorporate import functionality into a batch file. Cannot be used with **-e**, **-u**, or **-v**. Before importing, you should

Defining Your Synergy Methods

The Method Definition Utility

verify that no one else is using the SMC files. If you want to import only selected interfaces or methods from the XML file, use the Import Methods menu option; see [“Importing and Exporting Methods” on page 2-39](#).



When using the **-i** or **-u** option to update an existing SMC, pay close attention to the SMC you are updating, as these options overwrite data in the target SMC. You may want to make a habit of always specifying the SMC path on the command line when using these options.

-u *XMLfilename*

(optional) Update an existing catalog with the interface definitions in the specified XML file. Specify the complete path or use a logical if the file is not in the current working directory. This option causes existing interfaces in the SMC to be replaced with the interface of the same name in the XML file. Any new interfaces in the XML file are added to the SMC.

This option imports only methods in named interfaces; methods that are in an unnamed interface are ignored. (An “unnamed interface” means that in the XML file, the method is nested within an interface tag, but that tag has no name property. In the MDU, such a method would have no interface assigned, as is the case with XFPL_REGCLEANUP and XFPL_LOG.)

The MDU user interface does not display with **-u**, so you can use this option to incorporate update functionality into a batch file. Cannot be used with **-e**, **-i**, or **-v**. Before updating, you should verify that no one else is using the SMC files.



You can use the **-i** or **-u** option to create a new SMC from definitions in an XML file. See [“To create new files by importing an XML file” on page 2-44](#).

-v

(optional) Verify and update the repository structure sizes in the SMC. This option compares the structure sizes in the SMC with those in the repository and updates the SMC as necessary without displaying the MDU user interface. Cannot be used with **-e**, **-i**, or **-u**. Also reports enumerations that are present in the SMC but not the repository. See [“Verifying Repository Structure Sizes and Enumerations” on page 2-42](#) for more information.

-l *logfile*

(optional) Specify a log file to be used for logging errors during an import or logging changes made during repository verification. Use only with -i, -u, or -v. If -l is not specified, the default logfile (**import_smc.log** or **verify_smc.log**) will be created in the current working directory on OpenVMS and in the location specified with the TEMP environment variable on Windows and UNIX. (Note that these log files are created only when there are actually errors, warnings, or changes to log.)

-?

(optional) Display a list of options for the utility.

Examples The first example below shows how to start the MDU using the default SMC files and repository files. This command will use the SMC files in the location specified by XFPL_SMCPATH. If XFPL_SMCPATH is not set, **mdu** will use the files in the location specified by DBLDIR. The logic used to determine the default repository files is explained under the *rps_path* argument on [page 2-50](#).

```
dbr DBLDIR:mdu
```

To start the MDU using SMC files in the work\smc directory and the **rpsmain.ism** and **rpstext.ism** files in the work\rps directory, use this command line:

```
dbr DBLDIR:mdu c:\work\smc -r c:\work\rps
```

To start the MDU using SMC files in the work\smc directory and specify the repository files by name, use this command line:

```
dbr DBLDIR:mdu c:\work\smc -m c:\work\rps\MyMain.ism  
-t c:\work\rps\MyText.ism
```

The two examples below show how to start the MDU using SMC files on a remote machine that is running *x/Server*. You can use the full path to specify the location of the files or you can use a logical. We recommend the latter. If you do specify the path, you must use the correct path syntax (delimiters) for the operating system where the files are located—not the path syntax for the local system. The first example shows how to access files on a UNIX system named “elmo” using a path; the second shows how to access the files using a logical, which is defined on the target machine.

```
dbr DBLDIR:mdu /usr/SMCllocation/@elmo
```

```
dbr DBLDIR:mdu XFPL_SMCPATH:@elmo
```

Defining Your Synergy Methods

The Method Definition Utility

To start the MDU using repository files on a remote machine, you can specify the path or a logical, followed by the node name as shown in this example:

```
dbr DBLDIR:mdu XFPL_SMCPATH:@elmo -m RPSMFIL:@elmo
-t RPSTFIL:@elmo
```

To start the MDU on OpenVMS, you must define MDU as a foreign command and then execute it. In the example below, the MDU will look for the files **rpsmain.ism** and **rpstext.ism** in the DEV:[work] directory.

```
$ MDU==$DBLDIR:MDU
$ MDU -R DEV:[work]
```

The next three examples show how to use the **-e**, **-i**, and **-u** options. The first example exports the contents of the SMC file located in c:\work\smc to a file named **CatalogA.xml**. We have specified the repository location so that the correct structure and enumeration definitions are exported.

```
dbr DBLDIR:mdu c:\work\smc -r c:\work\rps
-e c:\temp\CatalogA.xml
```

This example imports the definitions in **CatalogA.xml** into the SMC located in d:\synergy\smc, replacing the entire catalog in that location. If there are any errors, they will be recorded in **myImpLog.txt**, and the import will be cancelled.

```
dbr DBLDIR:mdu d:\synergy\smc -i c:\temp\CatalogA.xml
-l c:\temp\myImpLog.txt
```

This example updates the definitions in the SMC located in d:\synergy\smc with the definitions in **CatalogB.xml**; interfaces of the same name will be replaced and new interfaces will be added.

```
dbr DBLDIR:mdu d:\synergy\smc -u c:\temp\CatalogB.xml
-l c:\temp\myImpLog.txt
```

The example below shows how to run the Verify Catalog utility, specifying the SMC and repository file locations. All changes made, as well as any problems encountered, will be logged to **myVLog.txt**.

```
dbr DBLDIR:mdu c:\work\smc -r c:\work\rps -v
-l c:\temp\myVLog.txt
```

The SMC/ELB Comparison Utility

The SMC/ELB Comparison utility (**smc_elb.exe**) reads methods and their associated ELBs (or shared images) in the SMC, and then verifies that there is an ELB containing those methods. This utility can be useful at deployment time to help ensure that you have all the necessary files. **Smc_elb.exe** is installed in the `dbl\bin` directory.



The SMC/ELB Comparison utility checks only for the ELBs that are referenced in the SMC. It does not check for any dependent ELBs that may be linked to the ELBs in the SMC.

If you're using Workbench, the SMC/ELB Comparison utility runs automatically when you generate Java class wrappers, create a COM type library, or generate C# classes. You can also run it independently in Workbench or from the command line on any platform. Running a comparison from the command line enables you to specify which SMC, interface, and method ID you want to check and how you want to receive the output (see below for Windows and UNIX instructions; see [page 2-57](#) for OpenVMS). Running a comparison from Workbench enables you to verify the interfaces selected in a particular project (see [page 2-58](#)).



If you used logicals in the SMC to point to the directories that your ELBs or shared images reside in, those logicals must be defined in the environment in order for **smc_elb** to be able to resolve them. **Smc_elb** does not read **xfpl.ini**, **SYNRC.COM**, or **DBLDIR:SERVER_INIT.COM**.

Windows and UNIX

Syntax `smc_elb [-s smc_path] [-i interface_name] [-m method_id]
[-o filename] [-e] [-b] [-l line_length] [-v] [-h]`

Arguments none

Run without any arguments, **smc_elb** will verify all methods in the SMC (in the location specified by DBLDIR) and display the output to the screen.

`-s smc_path`

(optional) The path where the SMC files are located. If not specified, **smc_elb** looks for the SMC files in the location specified by DBLDIR. If DBLDIR is not set, **smc_elb** looks in the current working directory.

Defining Your Synergy Methods

The SMC/ELB Comparison Utility

- i *interface_name*
(optional) Verify only the methods in the specified interface name. If you do not specify an interface name (or a method ID; see below), all methods in the SMC are verified.
- m *method_id*
(optional) A specific method ID you want to verify. If you specify both an interface name and a method ID, the interface name overrides the method ID and all methods in the interface are verified. If you do not specify a method ID (or an interface name; see above), all methods in the SMC are verified.
- o *filename*
(optional) File that you want the output written to. If the file does not exist, it is created. If it exists, it is overwritten. If **-o** is not specified, output is directed to the screen.
- e
(optional) Display all output to the screen while also writing it to a file. Use with **-o**.
- b
(optional) Display only errors to the screen or, if **-o** is specified, write only errors to a file. Success and informational messages are not displayed.
- l *line_length*
(optional) Number of characters per line for output. You can use this option when writing to a file (**-o** option) or the screen. The default line length is 255 characters, the maximum is 256, and the minimum is 20.
- v
(optional) Display the version number.
- h
(optional) Display a list of options.

OpenVMS

Syntax `smc_elb [/smc_path=smc_path] [/interface=interface_name]
[/method=method_id] [/output=filename] [/verbose] [/brief]
[/line_length=value] [/version]`

Arguments none

Run without any arguments, **smc_elb.exe** will verify all methods in the SMC (in the location specified by DBLDIR) and display the output to the screen.

`/smc_path=smc_path`

(optional) The path where the SMC files are located. If not specified, **smc_elb.exe** looks for the SMC files in the location specified by DBLDIR. If DBLDIR is not set, **smc_elb.exe** looks in the current working directory.

`/interface=interface_name`

(optional) Verify only the methods in the specified interface name. If you do not specify an interface name (or a method ID; see below), all methods in the SMC are verified.

`/method=method_id`

(optional) A specific method ID you want to verify. If you specify both an interface name and a method ID, the interface name overrides the method ID and all methods in the interface are verified. If you do not specify a method ID (or an interface name; see above), all methods in the SMC are verified.

`/output=filename`

(optional) File that you want the output written to. If the file does not exist, it is created. If it exists, a new file is created. If **/output** is not specified, output is directed to the screen.

`/verbose`

(optional) Display all output to the screen while also writing it to a file. Use with **/output**.

`/brief`

(optional) Display only errors to the screen or, if **/output** is specified, write only errors to a file. Success and informational messages are not displayed.

Defining Your Synergy Methods

The SMC/ELB Comparison Utility

`/line_length=value`

(optional) Number of characters per line for output. You can use this option when writing to a file (**/output** option) or the screen. The default line length is 255 characters, the maximum is 256, and the minimum is 20.

`/version`

(optional) Display the version number.

Running an SMC/ELB Comparison from Workbench

When you run the SMC/ELB Comparison utility from Workbench, it uses the SMC specified in the SMC directory field in the Component Information dialog box, and checks all methods in the interfaces that are selected in that dialog box.

1. Open a Java, COM, or .NET component project in Workbench.
2. Select Synergy/DE > Utilities > SMC/ELB Comparison. The utility runs and displays output in the Build tab of the Output toolbar.

Configuring and Running *xfServerPlus*

This chapter explains how to start, stop, and test *xfServerPlus*. It also includes information on the **xfpl.ini** file, using a log file, deploying *xfServerPlus*, debugging remote routines, and configuring *xfServerPlus* for remote data access.

The Big Picture

This section lists the steps required to configure and start *xfServerPlus*. For an overview of the entire process of developing a distributed application, see “The Big Picture” section for your client:

- ▶ Synergy, see [page 5-1](#)
 - ▶ Java, see [page 9-1](#)
 - ▶ COM, see [page 13-3](#)
 - ▶ .NET, see [page 16-3](#)
1. Create a user account on the *xfServerPlus* machine to run *xfServerPlus* sessions. For details on creating an account, see “[Running xfServerPlus on Windows](#)” on [page 3-3](#), “[Running xfServerPlus on UNIX](#)” on [page 3-9](#), or “[Running xfServerPlus on OpenVMS](#)” on [page 3-12](#). (Note: On OpenVMS, you must specify this account during installation. On Windows and UNIX, you can create the account after installation, if you like.)
 2. Install *xfServerPlus* on your Synergy server machine. The *xfServerPlus* installation includes the Method Definition Utility and the Synergy Method Catalog files.
 3. Add the necessary settings to the **xfpl.ini** file:
 - ▶ Set logging options for the *xfServerPlus* log. You can set options to turn logging on and off, specify the name of the logfile, specify single or multiple log files, and determine the type of information that is logged. See “[Setting Options for the xfServerPlus Log](#)” on [page 3-30](#).

Configuring and Running xfServerPlus

The Big Picture

- ▶ If you use logicals in the SMC to point to the directories that your ELBs or shared images reside in, define those logicals in the **xfpl.ini** file (Windows and UNIX) or **DBLDIR:SERVER_INIT.COM** (OpenVMS). See [“Defining Logicals” on page 1-4](#).
 - ▶ If you need to specify a base channel number for *xfServerPlus* to use when opening files, set **XFPL_BASECHAN** to the desired value. See [“Specifying a Base Channel Number” on page 1-7](#).
 - ▶ If you plan to use data compression, set **XFPL_COMPRESS** to ON. (Compression is supported for Java, COM, and .NET clients only.) See [“Configuring Compression” on page 3-22](#).
4. If you choose to put the **xfpl.ini** file somewhere other than **DBLDIR**, set the **XFPL_INIPATH** environment variable. See [“Setting the XFPL_INIPATH Environment Variable” on page 3-19](#).
 5. If you choose to put your Synergy Method Catalog somewhere other than **DBLDIR**, set the **XFPL_SMCPATH** environment variable so that *xfServerPlus* can find it. See [“Setting the XFPL_SMCPATH Environment Variable for xfServerPlus” on page 2-46](#).
 6. Start *xfServerPlus*. See [“Running xfServerPlus on Windows” on page 3-3](#), [“Running xfServerPlus on UNIX” on page 3-9](#), or [“Running xfServerPlus on OpenVMS” on page 3-12](#).

Running xfServerPlus

By default, xfServerPlus runs on port 2356.

Running xfServerPlus on Windows

We recommend that you set up an account with limited privileges specifically for running xfServerPlus sessions. This account can be a local account on the xfServerPlus machine or it can be a domain account. The account must have read/write permissions for any directories containing files that xfServerPlus or your Synergy methods will create or update, including the xfServerPlus logfile.

All clients will assume the persona of this account; consequently, it *should not* be a member of the administrator group. However, the user who registers and starts the xfServerPlus service *must* be a member of the administrator group.



The account that you create to run xfServerPlus sessions must allow a log-in on the machine that xfServerPlus is running on. If you cannot log on to the xfServerPlus machine using the account, verify that the account has the “log on locally” (or “allow log on locally” on some platforms) user right set. This is usually set by default for user-level accounts. However, if the xfServerPlus account is on a domain controller, and xfServerPlus is running on that domain controller, it is likely that this user right is *not* set by default.

You can register and start xfServerPlus either from the Synergy Configuration Program (see below) or from the command line (see [page 3-6](#)).

Starting xfServerPlus from the Synergy Configuration Program



Do not attempt to issue **rsynd** commands from the command line while the Synergy Configuration Program is running.

You must be a member of the administrator group to register and start xfServerPlus.

1. Start the Synergy Configuration Program (from the Start menu, select Programs > SynergyDE > Utilities > Synergy Configuration Program) and go to the xfServer/xfServerPlus tab.
2. Click the Add xfServerPlus Service button.

Configuring and Running xfServerPlus

Running xfServerPlus

3. Complete the fields in the xfServerPlus Information dialog box.

Service name. Enter the service name (registry key) for this service. The default is “xfspl”. This is the name that displays in the list of services on the xfServer/xfServerPlus tab.

Port number. Enter the port number for this service. Valid ports are in the range 1024 through 65534. The default is 2356.

Display name. Enter a display name for this service. This is the name that displays in the Windows Services console. If you leave this field blank, it defaults to “Synergy/DE xfServerPlus ####”, where #### is the port number.

Username. Enter the user name for the account that you created to run xfServerPlus sessions. This account can be either a local account or a domain account; it cannot be a member of the administrator group. If there is an account with the same user name on both the local machine and on a Windows domain, or on multiple domains, and you want to use a specific domain account, explicitly specify the domain in the format *user_name@domain_name* or *domain_name\user_name*.

Password. Enter the password for the account that you created to run xfServerPlus sessions.

Enable remote debugging. Check this box to enable remote debugging of this xfServerPlus service via a Telnet client. See [“Debugging Your Remote Synergy Routines” on page 3-40](#) for details.

Debug port number. Enter the port number that the debug server should listen on for the Telnet client. This port number must be different than the port that the xfServerPlus service listens on. Valid ports are in the range 1024 through 65534.

Timeout in seconds. Enter the number of seconds the server should wait for a connection from the Telnet client after the xfNetLink–xfServerPlus connection has been made. The default is 100 seconds; this value should be less than the connect time-out set on the client. (The connect time-out, which defaults to 120 seconds, is ‘B’ in [figure 5-1 on page 5-4](#).) If the Telnet client fails to connect within the specified time, the application will continue running normally, without debugging enabled.

Enable encryption. Select this option to enable encryption of data between *xfNetLink* and *xfServerPlus*, and then select the type of encryption desired, Master or Slave. See [“Using Encryption” on page 3-23](#) for details on this feature.



The encryption settings for the <Default> entry are used by both *xfServer* and *xfServerPlus*. For more information on the <Default> entry, see [“Using the <Default> entry”](#) in the “Configuring *xfServer*” chapter of the *Installation Configuration Guide*.

Certificate file. Specify the certificate file (.pem file) you created using either the full path or a logical. The default filename is **DBLDIR:rsynd.pem**, but you may choose another name and place the file anywhere you like.

Cipher type. Select the level of encryption desired: High, Medium, or Low. These three options map to specific cipher suites, which may vary depending on OpenSSL version. You can use the SLC option to the GETFA routine to see which cipher suite is being used. See [GETFA](#) in the “System-Supplied Subroutines, Functions, and Classes” chapter of the *Synergy Language Reference Manual*.

4. If you want to modify environment variables for this service (or for all services), click the Environment Setup button. From here, you can set or modify XFPL_INIPATH and XFPL_SMCPATH, as well as any other environment variables that you want *xfServerPlus* to use. For instructions, see [“Setting XFPL_INIPATH on Windows” on page 3-19](#) or [“Setting XFPL_SMCPATH on Windows” on page 2-46](#).



Environment variables set using the <Default> entry in the list of services apply to all *xfServerPlus* (and *xfServer*) services, but they can be overridden for individual services if desired. For more information on the <Default> entry, see [“Using the <Default> entry”](#) in the “Configuring *xfServer*” chapter of the *Installation Configuration Guide*.

5. Click OK in the *xfServerPlus* Information dialog box. The new service will display in the list of services.
6. Click Apply to register the service.
7. (optional) Click the Start Service button. See the table in [“Starting *xfServer*”](#) in the “Configuring *xfServer*” chapter of the *Installation Configuration Guide* for error and status codes that may display when starting or stopping *xfServerPlus*. If you

Configuring and Running xfServerPlus

Running xfServerPlus

don't want to start the service now, you can start it later from the Synergy Configuration Program, the Windows Services console, or the command line; see [step 2 on page 3-6](#).

Starting xfServerPlus from the command line



Do not attempt to issue **rsynd** commands from the command line while the Synergy Configuration Program is running.

You must be a member of the administrator group to register and start *xfServerPlus*. When you start *xfServerPlus* from the command line, the following defaults are used:

Port = 2356

Service name = *xfspl*

Display name = Synergy/DE *xfServerPlus*

You can override these defaults. See the syntax under “[Creating a second session](#)” on [page 3-7](#) for the options.

1. You must register the *xfServerPlus* service before starting it. At the command line, enter

```
rsynd -w -u xfspAcct/password [-p port] -r
```

You must supply a user name and password with the **-u** option when you register *xfServerPlus*. The password must be in clear text; it will be encoded in the registry.

If there is an account with the same user name on both the local machine and on a Windows domain, or on multiple domains, and you want to use a specific domain account, you must explicitly specify the domain name using the format *user_name@domain_name/password* or *domain_name\user_name/password*.

If *xfspAcct* is a member of the administrator group, *xfServerPlus* will return an error message and will not start.

2. Once *xfServerPlus* is registered, use one of the following methods to start it.

- At the command line, enter

```
net start serviceName
```

where *serviceName* is the name of the *xfServerPlus* service. The default is “*xfspl*”. If *xfServerPlus* starts successfully, you'll see a message to that effect.

- ▶ In the Windows Services console, select the service (the default display name is “Synergy/DE xfServerPlus”), and click the Start Service button or select Action > Start.
- ▶ In the Synergy Configuration Program, go to the xfServer/xfServerPlus tab, select the service and click the Start Service button.

See the table in “Starting xfServer” in the “Configuring xfServer” chapter of the *Installation Configuration Guide* for error and status codes that may display when starting or stopping xfServerPlus.

Creating a second session

You can create more than one instance of xfServerPlus by doing one of the following:

- ▶ From the Synergy Configuration Program, add another xfServerPlus session by repeating the instructions in “Starting xfServerPlus from the Synergy Configuration Program” on page 3-3. Each session must have a different service name, display name, and port number.
- ▶ From the command line, register another instance of xfServerPlus, specifying a different service name, display name, and port number. Use this syntax:

```
rsynd -w -u xfspAcct/password -p port -r -c serviceName  
-d "displayName"
```



For a complete list of options that can be used with **rsynd**, see “The rsynd Program” in the “Configuring xfServer” chapter of the *Installation Configuration Guide*.

Stopping xfServerPlus

You must be a member of the administrator group to stop *xfServerPlus*. When you stop *xfServerPlus*, existing connections are lost.

See the table in “[Starting xfServer](#)” in the “Configuring *xfServer*” chapter of the *Installation Configuration Guide* for error and status codes that may display when starting or stopping *xfServerPlus*.

► To stop *xfServerPlus* without unregistering it

Use one of these methods:

- In the Synergy Configuration Program, go to the *xfServer/xfServerPlus* tab, select the service, and click the Stop Service button.
- In the Windows Services console, select the service (the default display name is “Synergy/DE *xfServerPlus*”) and click the Stop Service button or select Action > Stop.

- At the command line, enter

```
net stop serviceName
```

where *serviceName* is the name of the *xfServerPlus* service. The default service name is “*xfspl*”.

- Use the **rsynd -w -q** command.

This stops the default *xfServerPlus*, **xfspl**, on the default port, 2356. If you are running *xfServerPlus* with a different service name, use the **-c** option to specify the service name. If you are running *xfServerPlus* on a non-default port, use the **-p** option to specify the port number.

► To stop *xfServerPlus* and unregister it

Use one of these methods:

- In the Synergy Configuration Program, go to the *xfServer/xfServerPlus* tab, select the service from the list, and click the Remove Service button. Click Yes at the confirmation prompt.
- Use the **rsynd -w -x** command.

This stops and unregisters the default *xfServerPlus*, **xfspl**, on the default port, 2356. If you are running *xfServerPlus* with a different service name, use the **-c** option to specify the service name. If you are running *xfServerPlus* on a non-default port, use the **-p** option to specify the port number.

Running xfServerPlus on UNIX

We recommend that you set up an account with limited privileges specifically for running xfServerPlus sessions. Use either a system-level environment variable for DBLDIR or add a DBLDIR entry to the **synrc** file in the user directory for the new account. Clients assume the persona of the user name that is specified or assumed during start-up; consequently, we recommend that you *not* give this account root access. You can start xfServerPlus either with or without a password.

Starting xfServerPlus with a password

This method of starting xfServerPlus requires that you supply the password for the account you created to run xfServerPlus sessions. The password must be encoded using the **setruser** utility; it cannot be entered in clear text. There are a couple of ways to specify a user password. For example:

```
rsynd -w -u `setruser`
```

This command launches the **setruser** utility, which prompts for a user name and password. Note that **setruser** must be enclosed between grave accent characters (```). After you enter the user name and password, the command starts **rsynd** with the specified user name and the encoded password on the default port (2356). If you want to use a different port, specify it with the **-p** option.

You can also run **setruser** to generate the encoded password string, and then include it in the start-up command. For example:

```
rsynd -w -u "username/\362\224c\261\351\224\374P"
```

Because this method does not require user input, you can put this command in a start-up file. Note that the user name/password string must be enclosed in double quotation marks; failure to include the quotation marks may result in a “wrong username/password” error. (For more information on **setruser**, see “[The setruser Utility](#)” in the “Configuring xfServer” chapter of the *Installation Configuration Guide*.)

Starting xfServerPlus without a password

This method enables you to start *xfServerPlus* from an authorized account without specifying the password for the account you created to run *xfServerPlus* sessions. (If the password is passed, it is ignored.) The syntax is

```
rsynd -w -u xfspAcct
```

where *xfspAcct* is the user name of the account you created to run *xfServerPlus* sessions. This starts *xfServerPlus* on the default port (2356). If you want to use a different port, specify it with the **-p** option. All clients assume the persona of *xfspAcct*.

This command can be executed by a user signed on as *xfspAcct* or by any user with root privileges (uid=0). However, if the user name itself (i.e., *xfspAcct*) is root, *xfServerPlus* will return an error and will not start. If *xfServerPlus* starts successfully, you'll see the message "All *xfServerPlus* clients will be run as user *xfspAcct*."

You can also start *xfServerPlus* without specifying a user name. The user name defaults to that of the user signed on. (Presumably, this is the account you created to run *xfServerPlus* sessions.) The syntax is

```
rsynd -w
```

This starts *xfServerPlus* on the default port (2356). If you want to use a different port, specify it with the **-p** option. All clients assume the persona of the user who started **rsynd**.

This command can be executed by any user *without* root authority. If start-up is successful, you'll see the message "All *xfServerPlus* clients will be run as user *userName*." If *userName* has root authority, *xfServerPlus* will return an error message and will not start.

Creating a second session

You can create more than one instance of *xfServerPlus* by specifying a different port for the additional session in the start-up syntax. (The default port is 2356.) For example:

```
rsynd -w -u xfspAcct -p port
```



For a complete list of options that can be used with **rsynd**, see "The **rsynd** Program" in the "Configuring *xfServer*" chapter of the *Installation Configuration Guide*.

Stopping xfServerPlus

There are two methods for stopping xfServerPlus (**rsynd**).

- To stop xfServerPlus without killing the existing connections, use the **-q** option:

```
rsynd -q -w
```

This stops xfServerPlus on the default port, 2356. If you are using a different port, omit the **-w** option and specify the port number with the **-p** option. For example:

```
rsynd -q -p 2445
```

This is the usual method for stopping xfServerPlus. Existing connections are allowed to continue, but new connections are blocked. Use this method when you need to start a new version or configuration of xfServerPlus on that port, or any time you want to prevent new access to the server without interrupting existing connections.

- To stop xfServerPlus *and kill all existing connections*, use the **-c** option with **-q**:

```
rsynd -q -c -w
```

This stops xfServerPlus on the default port, 2356. If you are using a different port, omit the **-w** option and specify the port number with the **-p** option.

All existing connections are terminated, new connections are blocked, and the service runtimes started by the server are terminated. Use this method only when you need exclusive access to the server.

When you stop the server in this manner, the error returned to the client will depend on which xfNetLink you are using.

- xfNetLink Synergy will return `$ERR_XFHALT` if there is a call in progress when the connection goes down. On subsequent attempts to connect while the connection remains down, xfNetLink Synergy will return `$ERR_XFNOCONN`.
- xfNetLink Java will throw `java.net.SocketException` if there is a call in progress when the connection goes down. (This could be a read or a write exception, depending on whether the client was sending or receiving at the moment of failure.) On subsequent attempts to connect while the connection remains down, xfNetLink Java will throw `java.net.ConnectException`.

Configuring and Running xfServerPlus

Running xfServerPlus

- ▶ *xfNetLink* COM and *xfNetLink* .NET will throw a socket error (if the client is sending) or a signal trap error (if the client is receiving) if there is a call in progress when the connection goes down. On subsequent attempts to connect while the connection remains down, *xfNetLink* COM and *xfNetLink* .NET will throw a connection refused error.

Running xfServerPlus on OpenVMS

We recommend that you set up an account with limited privileges specifically for running *xfServerPlus* sessions. You must specify this account during installation.

The account used to run *xfServerPlus* sessions must have the SHARE privilege. This account cannot have the privileges listed below, unless the /ALLOW_PRIVILEGED qualifier is specified in the start-up command. We recommend that you *not* allow these privileges because they give end-users privileges at the system administrator level.

ALTPRI	DOWNGRADE	SYSNAM
BYPASS	EXQUOTA	SYSPRV
CMEXEC	READALL	VOLPRO
CMKRNL	SECURITY	WORLD
DETACH	SETPRV	

After you install *xfServerPlus*, the *xfServerPlus* entry in the **SY\$MANAGER:SYNERGY_STARTUP.COM** file will include the following settings (you will see other settings in addition to these):

```
$ SYNERGY_SERVER -  
  /PORT=2356 -  
  /XFPL_FREE_POOL=2 -  
  /XFPL_ENABLE=xfspAcct
```

where

/PORT is the port that *xfServerPlus* will run on (default is 2356).

/XFPL_FREE_POOL is the number of remote execution sessions that will be kept available for user connections (default is 2; minimum is 1). Sessions remain in the pool until **rsynd** is shut down.

/XFPL_ENABLE enables *xfServerPlus*. *xfspAcct* is the user name of the account that the remote execution sessions will run under. (This is the account that was specified during installation.)

For information on the other xfServerPlus settings in the **SYNERGY_STARTUP.COM** file, see “[The rsynd Program](#)” in the “Configuring xfServer” chapter of the *Installation Configuration Guide*.

Once installed, xfServerPlus starts up as part of the machine start-up. If the user name specified has invalid privileges (see list above), xfServerPlus is not started, and the following error message is logged in the **rsynd** logfile. By default this file is named **node_rsynd_port.log** and located in DBLDIR.

Message from user SYSTEM on *MachineName* RSDMS\$MGR_2356: Security check: attempt to start xfServerPlus with elevated privilege failed.

To start xfServerPlus manually, use the command

```
$ RSYND /XFPL_ENABLE=xfspAcct
```

where *xfspAcct* is the user name of the account that xfServerPlus sessions will run under.



For a complete list of options for **rsynd**, additional information on starting **rsynd**, and details on **rsynd** logging, see the “[Configuring xfServer](#)” chapter in the *Installation Configuration Guide*.

You can check the status of xfServerPlus with the **servstat** program. Run option 9, Display xfServerPlus status. It will tell you the port on which xfServerPlus is enabled, the number of processes in use, and so on. Note that **servstat** does not report specific errors. If option 9 shows that xfServerPlus is not enabled, refer to the above-mentioned log file for information about what went wrong. See “[The servstat Program](#)” in the “General Utilities” chapter of *Synergy Language Tools* for details.

Creating a second session

You can create more than one instance of xfServerPlus by copying the xfServerPlus start-up section in the **SYNERGY_STARTUP.COM** file and adjusting the parameters as desired. Each instance of xfServerPlus must have a unique port. You will also probably want to specify a separate log file for each instance with the **/OUTPUT** qualifier. (Note that this log file is for the **rsynd** log, not the xfServerPlus log. See “[Logging](#)” in the OpenVMS section of the “Configuring xfServer” chapter of the *Installation Configuration Guide* for more information on **rsynd** logging.)

Configuring and Running xfServerPlus

Running xfServerPlus

If you want to be able to use separate **XFPL.INI** files with each instance of *xfServerPlus*, you must create a separate account for each instance and specify it with the `/XFPL_ENABLE` qualifier. See also [“Setting XFPL_INIPATH on OpenVMS” on page 3-21](#).



You can reduce the disk I/O overhead required for reading the SMC by installing a RAM drive and copying the SMC files and the **XFPL.INI** file to it. Then, set the `XFPL_INIPATH` and `XFPL_SMCPATH` logicals to point to the RAM drive. This will also reduce the time it takes to start the remote execution sessions.

Stopping xfServerPlus

Use one of these methods:

- ▶ While logged into the **SYSTEM** account, execute the command

```
$ RSYND/SHUTDOWN/PORT=port
```

where *port* specifies the port number that *xfServerPlus* is running on. Existing connections are not affected by the server system shutdown. New connections are not accepted after this command has executed.

- ▶ Use the **servstat** program. See [“The servstat Program”](#) in the “General Utilities” chapter of *Synergy Language Tools* for instructions.



If you need to refresh the pool without shutting down *xfServerPlus*, use **servstat** option 10, Purge *xfServerPlus* free pool. All processes in the *xfServerPlus* free pool will be destroyed and the pool will be repopulated with new processes. This option is especially useful when the **XFPL.INI** file has been modified, as it enables you to ensure that all processes in the free pool get the new settings.

Testing xfServerPlus

The **xfsplstst** program checks that *xfServerPlus* is running properly. The **xfsplstst** program makes calls to a test ELB or shared image named **xfpl_tst**, which is distributed with *xfServerPlus*. There are entries in the SMC for use by the test programs. (These are the methods in the *xfTest* interface in the distributed SMC.) If the ELB or any of the methods are missing, the tests will fail.



If the methods in the *xfTest* interface are not present in your SMC, you can import them from the **defaultsmc.xml** file. See [“Importing and Exporting Methods” on page 2-39](#).

There are also test programs for the *xfNetLink* clients. If you run one of the *xfNetLink* test programs and it fails, running **xfsplstst** can tell you if the problem is on the *xfServerPlus* side or the *xfNetLink* side. Refer to the section for your *xfNetLink* client:

- ▶ Synergy, see [page 5-6](#)
- ▶ Java, see [page 11-9](#)
- ▶ COM, see [page 15-11](#)
- ▶ .NET, see [page 18-9](#)

▶ To run the **xfsplstst** program

1. Make sure *xfServerPlus* has been started.
2. For Windows and UNIX, on the machine running *xfServerPlus*, enter

```
dbr DBLDIR:xfsplstst hostName hostPort
```

where *hostName* is the name of the *xfServerPlus* machine and *hostPort* is the port that *xfServerPlus* is running on.

On OpenVMS, you must define **xfsplstst** as a foreign command and then execute it. On the machine running *xfServerPlus*, enter

```
$ XFSPLTST:==$DBLDIR:XFSPLTST  
$ XFSPLTST hostName hostPort
```

As the test runs, information is printed to the screen and written to the **xfsplstst.log** file, located in the directory from which you ran the test. If any tests were unsuccessful, check **xfsplstst.log** for more information; the error message should

Configuring and Running xfServerPlus

xfServerPlus Status Codes

give you a clue as to what the problem is. If you cannot resolve the problem, call Synergy/DE Developer Support. Be sure to save the **xfsplstst.log** file; your Support engineer needs the information in this file to help you.

xfServerPlus Status Codes

These codes are returned to the client by *xfServerPlus* when it cannot start a remote execution session. Note that the meaning of the code may vary depending on the operating system that *xfServerPlus* is running on.

xfServerPlus Status Codes			
Operating system	Code	Description	What to do
Windows	1	Invalid user name.	Verify the user name and password used to run <i>xfServerPlus</i> sessions. Try logging on using that account. See “Running xfServerPlus on Windows” on page 3-3 .
	2	Cannot log on as user name.	
	3	Cannot impersonate as user name.	
	4	Cannot launch db s process.	Ensure that db s.exe exists on the <i>xfServerPlus</i> machine.
	5	Insufficient memory on server during start-up.	Upgrade memory or decrease the number of running processes.
	6	DBLDIR is not set.	Use the Synergy Configuration Program to set DBLDIR on the <i>xfServerPlus</i> machine.
	8	<i>xfNetLink</i> client version incompatible with server version.	Upgrade the client or server to the higher version. Running a newer client with an older server is not a supported configuration.
	9	No Synergy Runtime license.	Install a Runtime license on the <i>xfServerPlus</i> machine.
	38	Rsynd is running but <i>xfServerPlus</i> is not enabled.	Start rsynd with the -w option. See “Running xfServerPlus on Windows” on page 3-3 .
	176 - 200	Rsynd licensing error.	Check the Windows application event log on the <i>xfServerPlus</i> machine for details on the exact error.

xfServerPlus Status Codes			
Operating system	Code	Description	What to do
UNIX	1	Invalid user name or password.	Verify the user name and password used to run xfServerPlus sessions. See "Running xfServerPlus on UNIX" on page 3-9.
	2	Invalid user name.	
	3	Cannot fork child process.	See your system administrator.
	4	Cannot launch db s process.	Ensure that db s.exe exists on the xfServerPlus machine.
	6	DBLDIR is not set.	Set DBLDIR on the xfServerPlus machine.
	8	xfNetLink client version incompatible with server version.	Upgrade the client or server to the higher version. Running a newer client with an older server is not a supported configuration.
	9	No Synergy Runtime license.	Install a Runtime license on the xfServerPlus machine.
	38	Rsynd is running but xfServerPlus is not enabled.	Start rsynd with the -w option. See "Running xfServerPlus on UNIX" on page 3-9.
	176 - 200	Rsynd licensing error.	Check syslog on the xfServerPlus machine for details on the exact error.
OpenVMS	1	xfServerPlus is not licensed.	Purchase a license.
	2	xfServerPlus license limit exceeded.	Upgrade the number of xfServerPlus licenses.
	6	DBLDIR is not set.	Set DBLDIR on the xfServerPlus machine.
	7	The 14-day demo period or an extended demo period has expired.	Call your Synergy/DE customer service representative.
	8	xfNetLink client version incompatible with server version.	Upgrade the client or server to the higher version. Running a newer client with an older server is not a supported configuration.
	38	Rsynd is running but xfServerPlus is not enabled.	Start rsynd with the /XFPL_ENABLE option. See "Running xfServerPlus on OpenVMS" on page 3-12.

Using the xfpl.ini File

The **xfpl.ini** file is read by **xfpl.dbr** each time an *xfServerPlus* session is started. This file contains the settings that are used to specify logging options, compression, the base channel number for opening files, and—on Windows and UNIX—the logicals that point to your ELBs.

The settings in the **xfpl.ini** file, together with the settings in the **synrc** file or registry entry, create the environment that *xfServerPlus* runs in.

The default location for **xfpl.ini** is the DBLDIR directory. If you want to place the **xfpl.ini** file in a different location, you will need to set the XFPL_INIPATH environment variable to point to that location (see [page 3-19](#)).

If *xfServerPlus* encounters errors while reading the **xfpl.ini** file, they will be logged and then the connection will be terminated. By default, these errors are logged to the application event log (Windows), syslog (UNIX), or operator console (OpenVMS). If *xfServerPlus* logging is turned on (see [page 3-30](#)), they will also be recorded in the *xfServerPlus* log (**xfpl.log** by default).



See [Appendix A](#) for a complete list of the **xfpl.ini** file configuration settings.

Using an Alternate xfpl.ini File

There may be times when it would be convenient to use an alternate **xfpl.ini** file. At runtime, each instance of *xfServerPlus* can reference only a single **xfpl.ini** file; however, if you have multiple instances of *xfServerPlus* running on different ports, they can each reference a separate **xfpl.ini** file if desired.

To set up your system to use an alternate **xfpl.ini**, you need to

- ▶ decide on a location and create or copy the **xfpl.ini** file there.
- ▶ set the XFPL_INIPATH environment variable.

Setting the XFPL_INIPATH Environment Variable

Setting XFPL_INIPATH on Windows

On Windows, use the Synergy Configuration Program to set environment variables for *xfServerPlus* in the registry. You can set XFPL_INIPATH for all instances of *xfServerPlus* or for a specific instance of *xfServerPlus*.

► To set XFPL_INIPATH for all instances of *xfServerPlus*

This procedure sets XFPL_INIPATH in the Windows registry under HKEY_LOCAL_MACHINE\SOFTWARE\Synergex\Synergy xfServer\Synrc.

1. Start the Synergy Configuration Program (from the Start menu, select Programs > SynergyDE > Utilities > Synergy Configuration Program) and go to the xfServer/xfServerPlus tab.
2. Select Default from the list of services, click the Modify Service button, and then click the Environment Settings button.
3. Click the Add button that is grouped with the “Settings for all services” list.
4. Type the variable name (XFPL_INIPATH) and value in the Add Environment Setting dialog box, and click OK.
5. Click OK in the xfServer Information dialog box, and then click Apply in the Synergy Configuration Program.
6. You’ll be prompted to stop and restart all services so that the new settings take effect. Click Yes.

► To set XFPL_INIPATH for a specific instance of *xfServerPlus*

This procedure sets XFPL_INIPATH in the Windows registry under HKEY_LOCAL_MACHINE\SOFTWARE\Synergex\Synergy xfServer\ *serviceName*\Synrc. This setting overrides the “all instances” setting for a specific instance of *xfServerPlus*.

1. Start the Synergy Configuration Program (from the Start menu, select Programs > SynergyDE > Utilities > Synergy Configuration Program) and go to the xfServer/xfServerPlus tab.
2. Select the service from the list, click the Modify Service button, and then click the Environment Settings button.
3. Click the Add button that is grouped with the “Settings for *service name*” list.

Configuring and Running xfServerPlus

Using the xfpl.ini File

4. Type the variable name (XFPL_INIPATH) and value in the Add Environment Setting dialog box, and click OK.
5. Click OK in the xfServerPlus Information dialog box, and then click Apply in the Synergy Configuration Program. The service will be stopped and restarted.

Setting XFPL_INIPATH on UNIX

On UNIX, the *xfServerPlus* service reads settings from the environment and from the **synrc** file. There are **synrc** files at the system level (**/etc/synrc**) and at the user level (**/usr/username/.synrc**). At the system level, *xfServerPlus* supports both a generic **synrc** file and a port-specific **synrc** file. See below for details.



To include comments in the **synrc** file, precede the comment with a number sign (#).

- ▶ To specify the **xfpl.ini** file location for all instances of *xfServerPlus*, do one of the following:
 - ▶ Set XFPL_INIPATH as an environment variable.
 - ▶ Set XFPL_INIPATH in the generic **synrc** file (**/etc/synrc**). This setting overrides XFPL_INIPATH set in the environment.
- ▶ To specify the **xfpl.ini** file location for a specific instance of *xfServerPlus* by port number, create a file named **synrc.####**, where **####** is the port number that *xfServerPlus* is running on, and put it in the **/etc** directory. If you have multiple instances of *xfServerPlus* running on multiple ports, you can create a file for each instance and then set XFPL_INIPATH differently in each file. This setting overrides XFPL_INIPATH set either in the environment or in the generic **synrc** file.
- ▶ To specify the **xfpl.ini** file location for *xfServerPlus* by user name, set XFPL_INIPATH in the **.synrc** file for the user account that *xfServerPlus* sessions run under (**/usr/username/.synrc**). If you create several *xfServerPlus* accounts, each one could have a different **.synrc** file with different settings. Setting XFPL_INIPATH in the user's **.synrc** file overrides XFPL_INIPATH set in the environment, the generic **synrc** file, and the port-specific **synrc** file.

Setting XFPL_INIPATH on OpenVMS

XFPL_INIPATH should be set in **SYNRC.COM**. (Note that logicals defined in **SYNRC.COM** are case-sensitive.)



Although we recommend elsewhere in this manual that logicals be set in **DBLDIR:SERVER_INIT.COM**, you should not set XFPL_INIPATH in this file. **SERVER_INIT.COM** is invoked by a command in the **SYNERGY_STARTUP.COM** file *after* **rsynd** has started. This means that XFPL_INIPATH will not be executed until after the initial xfServerPlus sessions are started. In contrast, **SYNRC.COM** is read and processed each time an xfServerPlus session is started. Although using **SYNRC.COM** results in slower start-up, placing XFPL_INIPATH in **SYNRC.COM** is the only way to ensure that all xfServerPlus sessions are using the correct **XFPL.INI** file. For additional information on defining **rsynd** logicals on OpenVMS, see [“Defining logical names”](#) in the “Configuring xfServer” chapter of the *Installation Configuration Guide*.

- To specify the **XFPL.INI** file location for *all instances* of xfServerPlus, add this line to the **DBLDIR:SYNRC.COM** file:

```
$ DEFINE XFPL_INIPATH altdir
```

where *altdir* is the directory in which the **XFPL.INI** file is located.

- To specify the **XFPL.INI** file location for a *specific instance* of xfServerPlus, you must create a separate account for each instance, and then edit the **SYNRC.COM** file in that account’s directory, using the syntax given above. This will override the “all instances” setting. See [“Creating a second session” on page 3-13](#) for more information on running multiple instances of xfServerPlus on OpenVMS.

Configuring Compression

If your distributed application sends or receives data that contains repeated zeroes or spaces, turning on xfServerPlus compression will improve network throughput. Both sent and received data will be compressed.



xfServerPlus compression does not compress the entire packet; it compresses only repeated zeroes and spaces within the packet. (There must be at least three consecutive zeroes or spaces for compression to take place.) Consequently, if your distributed application passes only small amounts of data or passes large amounts of data that do not contain repeated zeroes or spaces, compression will be of no benefit and will likely degrade performance because the packets must be scanned. The improvement in network throughput for any particular application will depend on the average packet size and the amount of compressible data.

To turn on compression, set XFPL_COMPRESS in the **xfpl.ini** file to ON:

```
XFPL_COMPRESS=ON
```

Compression is supported only for version 8.1.5 and later Java, COM, and .NET clients used in conjunction with version 8.1.5 or later xfServerPlus. You do not need to do anything on the client side to enable compression. If compression is turned on in the **xfpl.ini** file and the client does not support compression, compression will simply not take place (no error will occur).

You can verify that compression is being used by checking the xfServerPlus log. If XFPL_SESS_INFO is set to ALL (see [page 3-31](#)) and packets are being compressed, you'll see "Compression = on" in the log. The average percentage of compression of all packets sent in each session will be logged as well, so that you may evaluate its benefit. (If you have debug logging turned on, the packets will be logged in their uncompressed form.)

To turn compression off, set XFPL_COMPRESS to OFF or remove the setting from the **xfpl.ini** file.

Using Encryption

The xfServerPlus encryption feature enables you to encrypt the transfer of sensitive data between an xfNetLink .NET or xfNetLink Synergy client and xfServerPlus. xfServerPlus interfaces with a third-party library, OpenSSL, to provide SSL support for secure data transport. Encryption is not supported for Java and COM clients.

To use encryption, both xfServerPlus and xfNetLink must be version 9.3 or higher. You have the option of using master or slave encryption. When *master* encryption is enabled, parameter and return value data for *all* methods is encrypted. When *slave* encryption is enabled, parameter and return value data for *selected* methods is encrypted. See [“Specifying the Data to Encrypt for Slave Encryption” on page 3-28](#) for details.

To implement encryption, you must start rsynd with the **-encrypt** option (/ENCRYPT on OpenVMS) or by selecting the Enable encryption option in the Synergy Configuration Program on Windows. You also specify a certificate file and the cipher (i.e., the level of encryption) to use. The three cipher options (high, medium, low) map to specific cipher suites, which may vary by OpenSSL version. See [“Setting up the xfServerPlus Machine for Encryption” on page 3-24](#) for details.

When encryption is enabled, you'll see an entry in the xfServerPlus log. If XFPL_SESS_INFO is set to ALL (see [page 3-31](#)), you'll see “Encryption = master” or “Encryption = slave” in the log, along with the level of encryption (high, medium, low). For slave encryption, if XFPL_FUNC_INFO is set to ALL, you will see “Encryption enabled” in the logfile for methods that are using encryption. When encryption is enabled, a string of 10 asterisks will display in the log file in place of the parameter, return value, and packet data for encrypted methods. Consequently, you may want to enable encryption only after you have thoroughly tested your application and have no need to examine packet data. (Note that once you have implemented encryption in your application, you cannot easily disable it; you would have to alter both the SMC and the client code in addition to turning off encryption on the server to return your application to an unencrypted state.)



If you see the error message “Cannot load random state”, it means there is not enough random data on the system to seed cryptographic algorithms. To correct this, you must define the SYNSSL RAND environment variable on either the server or the client (depending on where the error occurred) to point to a file that can be used to gather random data. (See [SYNSSL RAND](#) in the “Environment Variables” chapter of *Environment Variables and System Options* for more information.)

Setting up the xfServerPlus Machine for Encryption

1. Obtain and install OpenSSL.

For details on which version of OpenSSL is required for your operating system, see “[OpenSSL Requirements](#)” in the “Requirements and Considerations” chapter of the *Installation Configuration Manual*.

See Synergex KnowledgeBase article [100001979](#) for the latest information on where to obtain OpenSSL for various OSs and details on installation.

- ▶ For Windows, go to <http://www.openssl.org/related/binaries.html> and follow the link to download the version for your platform.



On Windows, we recommend that you install OpenSSL locally on the xfServerPlus machine so that it can be accessed by the xfServerPlus account. If OpenSSL is located on a remote machine, you must ensure that the xfServerPlus account can access that machine without being prompted for a logon. Setting up the xfServerPlus account as a domain account (rather than a local account) may be sufficient to allow access, depending on your network configuration. If you choose to use a domain account, see the note on [page 3-3](#).

- ▶ For UNIX, obtain the OpenSSL security kit specific to your operating system and version from your operating system vendor, if one is available. For some platforms, you may have to download and build the shared libraries. See Synergex KnowledgeBase article [100001979](#) for details.
- ▶ For OpenVMS, use the operating system-supplied libraries, HP SSL, available from http://h71000.www7.hp.com/openvms/products/ssl/ssl_download.html.

2. Ensure that the OpenSSL shared libraries are in the correct location or have been added to the correct path.

The library path must be set *before* registering **rsynd** on Windows or starting **rsynd** on UNIX and OpenVMS.

- ▶ On Windows, use the Synergy Configuration Program to set the OPENSSLIB environment variable to the location of the OpenSSL libraries. If the files are on a remote machine, use a UNC path rather than a mapped drive to specify the location. If OPENSSLIB is not set or the OpenSSL files are not found there, **rsynd** uses PATH to locate the OpenSSL libraries. If the libraries cannot be found, the operating system generates the error “Encryption is required but not available. A service specific error occurred: 14”.

- ▶ On most UNIX platforms, **rsynd** looks in the /usr/lib directory for the libraries. If they are not found there, it checks /lib. On the following platforms, the specified directory is checked first, followed by /usr/lib and /lib:
 - ▶ HP-UX 64-bit PA-RISC (309): /usr/lib/pa20_64
 - ▶ HP-UX 64-bit Itanium (509): /usr/lib/hpux64
 - ▶ IBM AIX 64-bit (304): /usr/lib64
 - ▶ Sun Solaris 64-bit SPARC (320): /usr/lib/64
 - ▶ Sun Solaris 64-bit i386 (420): /usr/lib/amd64
 - ▶ Linux 64-bit (428): /lib64

If the libraries cannot be found, the operating system generates the error “synssllib.so not available”.

- ▶ On OpenVMS, during installation, SYNSSLIB is set in SYS\$MANAGER:SYNERGY_STARTUP.COM to the full path and filename of **synssllib.exe**, the SSL runtime support file. If **synssllib.exe** cannot be found, **rsynd** generates the error “SYNSSLIB not set”.

3. Create a certificate file (.pem file).

If you name this file **rsynd.pem** and put it in DBLDIR, it will be used by default when you start **rsynd** with encryption enabled. However, you may name the file anything you like and put it elsewhere if desired, and then specify it with the **-cert** option (/CERTIFICATE on OpenVMS) or with the Synergy Configuration Program on Windows.

On Windows and OpenVMS, the certificate file cannot include a pass phrase. On UNIX, a pass phrase is permitted.



See [“Making a certificate request”](#) in the “Synergy HTTP Document Transport API” chapter of the *Synergy Language Reference Manual* for instructions on creating a certificate request file and sending it to a public certificate authority (CA). The CA will then send back a certificate file. For testing purposes, a local CA with self-signed certificates may suffice. See [“Testing your HTTPS setup locally”](#) in the above-mentioned chapter for steps on how to create a local file.

4. Start rsynd with encryption enabled.

Start **rsynd** with the **-encrypt** option (/ENCRYPT on OpenVMS) and specify master encryption if desired. (The default is slave.) Or, on Windows, start **rsynd** from the Synergy Configuration Program and select the Enable encryption option. Specify a non-default certificate filename and cipher if desired. See [“The rsynd Program”](#) in the “Configuring xfServer” chapter of the *Installation Configuration Guide* for detailed syntax and examples and [“Starting xfServerPlus from the Synergy Configuration Program”](#) on page 3-3.

Setting up the xfNetLink Machine for Encryption

xfNetLink Synergy

Install and configure OpenSSL on the *xfNetLink* Synergy machine.

- ▶ On Windows machines, if OpenSSL is installed in the default location (Windows\System32 directory), you don't need to set anything. If you install to another location, set OPENSSLIB to the location of the libraries. If OPENSSLIB is not set, PATH is used to find the OpenSSL libraries.
- ▶ On UNIX machines, the library path is used to find the OpenSSL libraries. This is LIBPATH on AIX, SHLIB_PATH on HP-UX, and LD_LIBRARY_PATH on all other systems. For 64-bit systems, you can also use the server paths listed in [step 2 on page 3-25](#).

xfNetLink .NET

For Windows XP, there is nothing to install or configure. For Windows Server 2003, there is also nothing to install or configure, but read [Microsoft KB article 948963](#) to see if it applies to your situation.

For Vista/2008 and higher, there is nothing to install, but you must ensure that SSL is enabled and that the desired cipher suites have been specified by doing the following:

1. Run **gpedit.msc** from the Run dialog. This launches the Local Group Policy Editor.
2. In the left panel, navigate to Computer Configuration > Administrative Templates > Network > SSL Configuration Settings.
3. In the right panel, double-click on SSL Cipher Suite Order to display the SSL Cipher Suite Order Properties dialog.
4. In the properties dialog, select Enabled on the Setting tab, and click the Apply button. You will see the enabled suites listed in the SSL Cipher Suites field. (The default is Not Configured, which does enable some cipher suites, but does not necessarily ensure that you have the correct suites enabled.)
5. Go to the Explain tab and follow the instructions to select the desired cipher suites. (Scroll down to the bottom of the window to see the numbered steps.) When you run xfServerPlus with encryption enabled, one of these suites will be selected based on the level of encryption (high, medium, low) you specified. You can see which suite is selected in the xfNetLink .NET logfile; see [“Using Client-Side Logging”](#) on page 18-7.
6. When you are done selecting cipher suites, click OK in the SSL Cipher Suite Order Properties dialog and then close the Local Group Policy Editor.

Specifying the Data to Encrypt for Slave Encryption

xfNetLink Synergy

There are two ways to mark methods for encryption when you are using xfNetLink Synergy.

- ▶ Use the **/encrypt** switch only. The **/encrypt** switch is appended to the end of the method ID argument in the RXSUBR or RX_SETRMTFNC routine. For example,

```
xcall rxsubr(netid, "mymethodid/encrypt", arg1, arg2)
```

If you use the **/encrypt** switch, you do not need to mark the method for encryption in the SMC, though you may do so if you like (see next bullet).

If you use the **/encrypt** switch, and encryption is not enabled on the xfServerPlus machine, the error \$ERR_XFSERVNOSEC, "Encryption not enabled on server", is returned. See [%RXSUBR on page 7-23](#) and [RX_SETRMTFNC on page 7-16](#) for more information about using **/encrypt**.

- ▶ Mark the method for encryption in the SMC and also use the **/encrypt** switch. Select the "Enable encryption" checkbox when defining the method in the MDU (or specify the encrypt=true property in the xfMethod attribute). If a method is marked for encryption in the SMC, you *must* also use the **/encrypt** switch as described above; else, the data will be sent unencrypted, and the error \$ERR_XFMETHCRYPT, "Method requires encryption", will be returned. Although it is more effort to mark methods for encryption in the SMC, the advantage is that an error will result if developers forget to include the **/encrypt** switch when they are coding.

xfNetLink .NET

To indicate which methods should be encrypted when you are using xfNetLink .NET, select the Enable encryption checkbox when defining the method in the MDU (or specify the encrypt=true property in the xfMethod attribute), and then generate classes and build the assembly.

When you generate classes, a new attribute named xfAttr, with the member encrypt=true, is added to each method that is marked for encryption in the SMC. If the client sends clear data when the method is marked for encryption in the SMC, the error "Method requires encryption" is generated.

Using Server-Side Logging

By default, *xfServerPlus* always logs errors to the application event log (Windows), syslog (UNIX), or operator console (OpenVMS). On Windows, the service runtime (**dfs.exe**) also logs traceback information from fatal errors to the application event log on the server machine.



On UNIX, syslog may not be running by default. See your system administrator to ensure the daemon is running.

On OpenVMS, set `REPLY/ENABLE=NETWORK` to send messages to the operator console.

You can log additional information with the *xfServerPlus* log, which is a record of remote access requests, data passed and returned, and errors. It can be used for troubleshooting and as a record of who has logged into your system. The log resides on the machine that *xfServerPlus* is running on. By default, *xfServerPlus* logging is turned off.

On Windows and UNIX, the default is to create a separate *xfServerPlus* log file for each session; you can specify a single log file if desired. See [“Setting Options for the xfServerPlus Log”](#) below for details on this and other options.

On OpenVMS, the *xfServerPlus* log produces a single log file for each instance of *xfServerPlus*. There is no option for multiple log files. You can use **servstat** option 11, Cycle *xfServerPlus* log file, to close and then open a new version of the log file. This enables you to examine the log file without shutting down **rsynd**. (See [“The servstat Program”](#) in the “General Utilities” chapter of *Synergy Language Tools* for details.)

When *xfServerPlus* logging is turned on, you can use the `XFPL_LOG` routine to make application-defined entries in the *xfServerPlus* log from either the server or the client application. See [XFPL_LOG on page 1-33](#) for more information.

Setting Options for the xfServerPlus Log

All logging options are set in the **xfpl.ini** file (see [page 3-18](#)). The logging options enable you to turn logging on and off, specify a name and location for the log file, indicate whether you want single or multiple log files, and control what data is logged.

For best results, we recommend that you put the XFPL_LOG setting *first* in the **xfpl.ini** file, followed by the other logging settings and then by the non-logging settings, such as XFPL_LOGICAL. This enables *xfServerPlus* to create the log file before processing the other settings. Any errors that *xfServerPlus* encounters while reading the **xfpl.ini** file will then be recorded in the *xfServerPlus* log file. Even after encountering an error in the **xfpl.ini** file, *xfServerPlus* will continue processing the file so that all errors can be logged before terminating the remote connection. (These errors will also be output to the application event log on Windows, syslog on UNIX, or operator console on OpenVMS.)



If you are using a single log file, it will become quite large and should be cleared out periodically. You can either delete the entire file or open the file in a text editor and selectively delete material that is no longer needed.

If you are using multiple log files (Windows and UNIX only), the files themselves will not become very large, but large numbers of them will accumulate on your server and may need periodically to be deleted.

XFPL_LOG

Turns logging on and off. Additional settings control the type of information that is logged.

Values: ON, OFF

Default value: OFF

Example: XFPL_LOG=ON

XFPL_LOGFILE

Points to the name and location of the log file. The log must be on the machine that *xfServerPlus* is running on. If logging is turned on and you do not specify a file name and location, the defaults are used. If you are generating multiple log files (XFPL_SINGLELOGFILE is set to OFF; see below), a date/time stamp will be appended to the end of the logfile name.

Default value: DBLDIR:xfpl.log

Example: XFPL_LOGFILE=c:\synergyde\my_log.log



(Windows) If no information is recorded in the log file, or if a log file is not created at all, verify that the account used to run xfServerPlus sessions has write permission for the directory in which the log is located.

xfServerPlus installs by default in c:\Program Files\Synergex\SynergyDE. If the account used to run xfServerPlus sessions was created with user level privileges, it will not be able to write to a log file in the default location because (by default) accounts in the user group cannot write to the Program Files directory or its subdirectories. Rather than change the privileges on the default directory, we recommend that you move the log file to another location.

XFPL_SINGLELOGFILE

(Windows and UNIX only) Determines whether information is logged to a single file or multiple files. By default, information from each session is logged to a separate log file named with the log file name plus a date and time stamp to differentiate the files. If you would prefer that all sessions write to a single log file, set XFPL_SINGLELOGFILE to ON. You can identify which log entries are associated with which session by the session ID.

One of the advantages of multi-file logging is improved performance. Because only one process is writing to the file, it can remain open. With single-file logging, the file must be opened and closed because different processes are writing to it, and this slows performance.

Values: ON, OFF

Default value: OFF

Example: XFPL_SINGLELOGFILE=ON

XFPL_SESS_INFO

Determines the level of session logging. Session logging records information about the connection, such as when the session was started and stopped and whether compression was enabled.

Values: NONE, CRITICAL, ALL

Default value: NONE

Example: XFPL_SESS_INFO=CRITICAL

Configuring and Running xfServerPlus

Using Server-Side Logging

The following table shows the information that is logged when XFPL_SESS_INFO is set to CRITICAL or ALL.

XFPL_SESS_INFO		
Value	Information logged	Description
CRITICAL or ALL	Session ID	ID that indicates the transactions related to the activities of a single end-user from log-on to log-off.
CRITICAL or ALL	xfServerPlusRemoteSession started	Date and time session started (log-on).
CRITICAL or ALL	xfNetLink IP address	IP address of the xfNetLink machine that called xfServerPlus.
CRITICAL or ALL	Domain	Domain name or IP address of the xfNetLink machine.
CRITICAL or ALL	xfServerPlusRemoteSession connected to	The type of client. This shows whether xfServerPlus is connected to a Synergy, Java, COM, or .NET client.
CRITICAL or ALL	Errors	Error packet that is sent to xfNetLink from xfServerPlus if an error is detected. Also shows the line number at which the error occurred.
CRITICAL or ALL	xfServerPlusRemoteSession stopped	Date and time session terminated (log-off).
ALL	xfNetLink port	Port on which xfNetLink is listening. Displays only for pre-8.3 client.
ALL	Server protocol version	Protocol version of xfServerPlus. See the table on page 12-19 for the correspondence between protocol and software versions.
ALL	Compression	Indicates whether xfServerPlus compression is on or off.
ALL	Average compression	The average percentage of compression for data sent and received during this session. (Displays only if compression is on.)

XFPL_SESS_INFO (continued)		
Value	Information logged	Description
ALL	Encryption	If encryption is enabled, displays the type of encryption: master or slave.
ALL	Cipher	If encryption is enabled, displays the level of encryption: high, medium, low.
ALL	xfServerPlusRemoteSession session time-out	If SET_XFPL_TIMEOUT is called, this shows the value that the time-out is set to. If xfServerPlus times out, that fact will be recorded in the log.



Because many users may access your application at the same time, if you are using a single log file, entries regarding a single session will likely be interspersed with entries from other sessions. The session ID at the beginning of each line enables you to identify entries by session.

The example below shows the type of information that is recorded when XFPL_SESS_INFO is set to CRITICAL. “0000539B” is the session ID.

```
000007F4: xfServerPlusRemoteSession 9.3.1 started: 20-AUG-2009
16:35:49
000007F4: xfNetLink ip address = 111.22.33.44, 6F16212C
000007F4: Domain: webtest Domain or IP
000007F4: xfServerPlusRemoteSession connected to .Net client
000007F4: ***** Error #2012 at line 652 *****
000007F4: 2012: Routine not found in ELB.
000007F4: 2012: Error occurred during lookup of external
routine.
000007F4: 2012: Routine name: function_fred
000007F4: 2012: ELB name: DBLDIR:xfpl_tst
000007F4: ***** End of Error *****
000007F4: xfServerPlusRemoteSession 9.3.1 stopped: 20-AUG-2009
16:35:53
```

Configuring and Running xfServerPlus

Using Server-Side Logging

The example below shows the type of information that is recorded when XFPL_SESS_INFO is set to ALL.

```
00000308: xfServerPlusRemoteSession 9.3.1 started: 20-AUG-2009
16:29:56
00000308: xfNetLink ip address = 111.22.33.44, 6F16212C
00000308: Server Protocol Version: 5
00000308: Domain: webtest Domain or IP
00000308: Compression = on
00000308: xfServerPlusRemoteSession connected to .Net client
00000308: xfServerPlusRemoteSession timeout set to 2 minutes and
7 seconds
00000308: Average Compression
00000308:   sends   : 11.18%
00000308:   receives: 0.00%
00000308: xfServerPlusRemoteSession 9.3.1 stopped: 20-AUG-2009
16:29:57
```

XFPL_FUNC_INFO

Determines the level of function and subroutine logging. Function logging records information about the Synergy routines that are called from the client.

Values: NONE, CRITICAL, ALL

Default value: NONE

Example: XFPL_FUNC_INFO=ALL

The following table shows the information that is logged when XFPL_FUNC_INFO is set to CRITICAL or ALL.

XFPL_FUNC_INFO		
Value	Information logged	Description
CRITICAL or ALL	Session ID	Indicates the transactions related to the activities of a single end-user from log-on to log-off. See the note on page 3-33 .
CRITICAL or ALL	Packet received at	Date and time that xfServerPlus received the packet.
CRITICAL or ALL	Method ID	Method ID that was called.
CRITICAL or ALL	Function	Name of the Synergy routine that was called.

XFPL_FUNC_INFO (continued)		
Value	Information logged	Description
CRITICAL or ALL	ELB	ELB or shared image in which the called routine is located.
CRITICAL or ALL	Packet returned at	Date and time the packet was returned by <i>xfServerPlus</i> .
ALL	Encryption enabled	If slave encryption is enabled, this indicates that the method is encrypted.
ALL	Parameter [n]	Data type, parameter size, and actual data passed for each parameter. [n] is the parameter sequence number; the data is enclosed in square brackets. If encryption is enabled, the log displays a string of 10 asterisks instead of the packet contents. The following abbreviations are used for data type: AL = alpha BI = binary (handle) DEC = decimal ENUM = enumeration HA = memory handle IMPDEC = implied-decimal INT = integer SS = System.String STR = structure
ALL	Function called at	Date and time the routine was called.
ALL	Function return value	The value returned by the function. If encryption is enabled, the log displays a string of 10 asterisks instead of the packet contents.
ALL	Returning parameter [n]	Data type, parameter size, and returned data for parameters flagged as having return data in the SMC. (These are "Out" or "In/Out" parameters). [n] is the parameter sequence number; the data is enclosed in square brackets. If encryption is enabled, the log displays a string of 10 asterisks instead of the packet contents. See above for data type abbreviations.

Configuring and Running xfServerPlus

Using Server-Side Logging

The example below shows the type of information that is recorded when XFPL_FUNC_INFO is set to CRITICAL. The final “packet received” message is the shutdown message received by *xfServerPlus* from the client.

```
00000418: Packet received at 20-AUG-2009 16:25:57
00000418: Method id: xfpl_tst1
00000418: Function: function_one
00000418: ELB: DBLDIR:xfpl_tst
00000418: Packet returned at 20-AUG-2009 16:25:57
00000418: Packet received at 20-AUG-2009 16:25:58
```

The example below shows the type of information that is recorded when XFPL_FUNC_INFO is set to ALL.

```
000003E0: Packet received at 20-AUG-2009 16:16:50
000003E0: Method id: xfpl_tst6
000003E0: Function: function_ten
000003E0: ELB: DBLDIR:xfpl_tst
000003E0: **** Incoming Parameters ****
000003E0: Parameter[1] = Type: (AL), Size: (5)
000003E0: Data [abcde]
000003E0: Parameter[2] = Type: (AL), Size: (5)
000003E0: Data [54321]
000003E0: **** End of Incoming Parameters ****
000003E0: Function called at 20-AUG-2009 16:16:50
000003E0: **** Outgoing Parameters ****
000003E0: Function Return Value: 123456789
000003E0: Returning Parameter[2] = Type: (AL), Size: (5)
000003E0: Data [back]
000003E0: **** End of Outgoing Parameters ****
000003E0: Packet returned at 20-AUG-2009 16:16:50
000003E0: Packet received at 20-AUG-2009 16:16:51
```

XFPL_DEBUG

Turns debug logging on and off. Debug logging records information that may be useful in troubleshooting *xfServerPlus* and *xfNetLink* errors.

Values: ON, OFF

Default: OFF

Example: XFPL_DEBUG=ON

The following table shows the information that is recorded when XFPL_DEBUG is set to ON.

XFPL_DEBUG	
Information logged	Description
Ini	Initialization. Shows what XFPL_SMCPATH is set to; if not set, the default is used (DBLDIR) and this line does not display.
Information that is recorded when XFPL_SESS_INFO is set to ALL	Session information; see page 3-32 .
Information that is recorded when XFPL_FUNC_INFO is set to ALL	Function information; see page 3-34 .
Command line	Command line used to start <i>xfServerPlus</i> .
Packet	Text of every packet received by <i>xfServerPlus</i> or returned to <i>xfNetLink</i> . For outgoing packets, the initial character in the packet indicates the type of <i>xfNetLink</i> client: C = COM client D = Synergy client J = Java client N = .NET client Returned packets are indicated by an 'R'. If encryption is enabled, the log displays a string of 10 asterisks instead of the packet contents.
Not returning parameter [n]	Data type and parameter size of parameters that were sent but are not being returned. (These are "In" parameters.) [n] is the parameter sequence number. The following abbreviations are used for data type: AL = alpha BI = binary (handle) DEC = decimal ENUM = enumeration HA = memory handle IMPDEC = implied-decimal INT = integer SS = System.String STR = structure The log also includes parameter and returning parameter information, recorded as part of the function information.

Configuring and Running xfServerPlus

Using Server-Side Logging

The example below shows the type of information that is recorded when XFPL_DEBUG is turned on.

```
00000D00: ini: XFPL_SMCPATH translates to {C:\work}
00000D00: xfServerPlusRemoteSession 9.3.1 started: 20-AUG-2009
12:40:13
00000D00: Command line = {C:\Synergyde\dbl\bin\dbs.exe -r
DBLDIR:xfpl 6F16212C 0}
00000D00: xfNetLink ip address = 111.22.33.44, 6F16212C
00000D00: Packet = {      }
00000D00: Server Protocol Version: 5
00000D00: Packet = {      20-AUG-2009      12:40:13;00      }
00000D00: Packet = {J0000webtest Domain or IP;Challenge
Response;      }
00000D00: Domain: webtest Domain or IP
00000D00: Compression = off
00000D00: xfServerPlusRemoteSession connected to .Net client
00000D00: Packet = {Nxfp1_tst6;2;AL5#abcde;AL5#54321;}
00000D00: Packet received at 20-AUG-2009 12:40:14
00000D00: Method id: xfp1_tst6
00000D00: Function: function_ten
00000D00: ELB: DBLDIR:xfpl_tst
00000D00: **** Incoming Parameters ****
00000D00: Parameter[1] = Type: (AL), Size: (5)
00000D00: Data [abcde]
00000D00: Parameter[2] = Type: (AL), Size: (5)
00000D00: Data [54321]
00000D00: **** End of Incoming Parameters ****
00000D00: Function function_ten called at 20-AUG-2009 12:40:14
00000D00: Function function_ten returned at 20-AUG-2009 12:40:14
00000D00: **** Outgoing Parameters ****
00000D00: Function Return Value: 123456789
00000D00: Not returning Parameter[1] = Type: (AL), Size: (5)
00000D00: Returning Parameter[2] = Type: (AL), Size: (5)
00000D00: Data [back]
00000D00: **** End of Outgoing Parameters ****
00000D00: Packet returned at 20-AUG-2009 12:40:14
00000D00: Packet =
{Rxfp1_tst6;002;000DE9#123456789;002AL4#back;}
00000D00: Packet = {S;0;;}
00000D00: Packet received at 20-AUG-2009 12:40:14
00000D00: xfServerPlusRemoteSession 9.3.1 stopped: 20-AUG-2009
12:40:14
```

Error Messages in the xfServerPlus Log

The table below lists the error numbers and the message text that appear in the xfServerPlus log. Errors recorded in the event log (Windows), syslog (UNIX), or operator console (OpenVMS) use these same error numbers, but the message text may be slightly different.

Errors in the 2000 series are logged and cause routine execution to stop, although xfServerPlus continues running. Errors in the 3000 series are logged but execution continues. The error message text shown in the table may be accompanied by additional detail information.

You may also see regular Synergy Language errors with additional detail information. See the “[Error Messages](#)” chapter in *Synergy Language Tools* for assistance with those errors.

xfServerPlus Error Messages	
Error number	Error message
2001	Incorrect message type
2002	Method ID too long
2003	Invalid number of parameters
2004	Parse error
2005	Invalid Synergy Type sent
2006	A required parameter was not passed
2007	Invalid data type specified in Method Catalog
2008	Error occurred in mapping array element
2009	Couldn't open file output channel
2010	Invalid operation type in file_read
2011	Method ID not found
2012	Routine <i>routine_name</i> not found in ELB <i>ELB_name</i>
2013	Unable to open method catalog file
2014	Unable to open method parameter file
2015	ELB not found

xfServerPlus Error Messages (continued)	
Error number	Error message
2016	Global not found: <i>global_name</i>
2017	File not found: <i>filename</i>
3001	Integer length in Synergy Method Catalog invalid
3002	Incoming data too big for the desired variable

Debugging Your Remote Synergy Routines

During normal operation, *xfServerPlus* runs as a background process (using the **dbS** service runtime) without support for console operations such as the Synergy debugger. This improves efficiency and minimizes memory requirements. However, you may at times need to debug the Synergy routines in the ELBs that are called from *xfServerPlus*. If you are running *xfServerPlus* on Windows or UNIX, there are two methods for doing this: running an *xfServerPlus* session in debug mode and debugging via Telnet. The latter is the recommended method. If you are running *xfServerPlus* on OpenVMS, there is only one debug method available: running an *xfServerPlus* session in debug mode.



(Windows) When debugging Synergy routines using either of the methods described below, **xfpl.dbr** uses the regular runtime (**dbr**) instead of the service runtime (**dbS**). This means that your environment may change, because **dbr** always reads the **synergy.ini** file, whereas **dbS** reads it only when SFWINIPATH is set. We recommend that you use SFWINIPATH to point to the location of your **synergy.ini** file and thereby avoid potential problems. For more information on **dbS**, see “The service runtimes” in the “Building and Running Synergy Language Programs” chapter of *Synergy Language Tools*.

Running an *xfServerPlus* session in debug mode

To run an *xfServerPlus* session in debug mode, you will need to alter your *xfNetLink* client code to call a special debug initialization method, start the client application, and then manually connect an *xfServerPlus* session to it. (On OpenVMS, you do not need to alter your client code.)

If your client is *xfNetLink* Synergy and you want to view packets on the client side, you must use this method. The Telnet method does not support this feature. (Note that you can always view packets on the server side in the *xfServerPlus* log.)

If you have a Java, COM, or .NET client that uses pooling, note that the pool will simply be ignored when you run an *xfServerPlus* session in debug mode, because your application will be using the session that you start manually. However, you should verify that the pool isn't using all available *xfServerPlus* licenses before you begin debugging.

The instructions for using this method vary slightly from one client to the next and are documented with each client. For information on this debugging method, refer to the “Running an *xfServerPlus* Session in Debug Mode” section for your *xfNetLink* client:

- ▶ Synergy, see [page 6-5](#)
- ▶ Java, see [page 11-11](#)
- ▶ COM, see [page 15-12](#)
- ▶ .NET, see [page 18-10](#)

Debugging via Telnet

To debug your remote routines via Telnet, you will first need to start *xfServerPlus* with remote debugging enabled. Starting **rsynd** in this manner causes it to start the **xfpl.dbr** session using the **dbr -rd** command. (See “[Debugging remotely](#)” in the “Debugging Your Synergy Programs” chapter of *Synergy Language Tools* for more information on running **dbr** with the **-rd** option.) Next, you'll start your *xfNetLink* client application. Once the *xfNetLink*–*xfServerPlus* connection is made, you'll access the server session via a Telnet session. The machine running the Telnet session may be the *xfServerPlus* machine, the *xfNetLink* client machine, or a separate machine. The primary advantage to this method is that it does not require you to modify your client code.

If there is a firewall between your Telnet client and *xfServerPlus*, the firewall must be configured to allow Telnet access on the debug port number. (Most firewalls are configured to prohibit Telnet access.) If you do not have authorization to reconfigure the firewall, you can either run Telnet on a machine within the firewall or use the other debug method.

See “[Debugging Remote Synergy Routines via Telnet](#)” below for instructions on using the Telnet method.

Debugging Remote Synergy Routines via Telnet

You can debug remote Synergy routines via Telnet when xfServerPlus is on Windows or UNIX. If you are running xfServerPlus on OpenVMS, see [“Running an xfServerPlus session in debug mode” on page 3-40](#).

If your Java, COM, or .NET client uses pooling, you should set both the minimum and maximum pool size to 1 before using Telnet debugging.

1. On your xfServerPlus machine, restart your xfServerPlus session with remote debugging enabled (or, you can start a completely new session on a different port if desired).

- On Windows, run the Synergy Configuration Program. On the xfServer/xfServerPlus tab, select the desired xfServerPlus service and click the Modify Service button. Select “Enable remote debugging”, specify a debug port number for the server to listen on for the Telnet client, and indicate a time-out value, if desired. The default is 100 seconds. This time-out measures how long the server will wait for a Telnet connection after the xfNetLink-xfServerPlus connection has been established. This time-out should always be less than the connect time-out set on the client, which defaults to 120 seconds. When you click the Apply button on the xfServer/xfServerPlus tab, xfServerPlus will be started with debug enabled.

For details on using the Synergy Configuration Program, see [“Starting xfServerPlus from the Synergy Configuration Program” on page 3-3](#) or refer to the Synergy Configuration Program online help.

- On UNIX, stop the current xfServerPlus service, and then restart it with the **-rd** option:

```
rsynd -rd debug_port[:timeout] -w -u xfspAcct
```

where *debug_port* is the port number that the xfServerPlus machine should listen on for the Telnet client, and *timeout* is the number of seconds that the server should wait for a Telnet connection after the xfNetLink-xfServerPlus connection has been made. The default is 100 seconds. This time-out measures how long the server will wait for a Telnet connection after the xfNetLink-xfServerPlus connection has been established. This time-out should always be less than the connect time-out set on the client, which defaults to 120 seconds. Include the other command line options that you normally use when starting xfServerPlus.

For more information about starting *xfServerPlus*, see “[Running *xfServerPlus* on UNIX](#)” on page 3-9. For complete **rsynd** syntax, see “[The *rsynd* Program](#)” in the “Configuring *xfServer*” chapter of the *Installation Configuration Guide*.

2. Run the *xfNetLink* client application so that a connection is made to *xfServerPlus*.
3. Start a Telnet session and connect to the *xfServerPlus* machine by specifying the server’s IP address (or “localhost” if you are running the Telnet session on the *xfServerPlus* machine) and the debug port that you specified in [step 1](#). You can run the Telnet session on any machine you like and use whatever Telnet application you prefer. The *xfServerPlus* machine becomes the “debug server”, and the Telnet session becomes the “debug client”. The debug prompt is sent to the Telnet session window.

Once the Telnet session has connected, the remote debug session works just like any other Synergy debug session. You will need to use the OPENELB debugger command to open the ELBs containing your Synergy routines before setting a breakpoint in one of those routines. Any ELBs linked to the opened ELB will also be opened.



Because the **xfpl.ini** file will not have been read at this point, you cannot use logicals in the file specification used with OPENELB; you must use the full path instead.

For general information about the Synergy debugger, see the “[Debugging Your Synergy Programs](#)” chapter in *Synergy Language Tools*. For details on the OPENELB command, see [OPENELB](#) in that same chapter.

4. When your distributed application finishes and the *xfServerPlus* connection to *xfNetLink* is closed, the Telnet session will also close. Optionally, you can use the QUIT or EXIT commands to close the debugger. Shutting down the Telnet session while in the midst of debugging will cause the application to continue running normally, without debugging enabled.
5. When you are through debugging, run the Synergy Configuration Program, or restart **rsynd** without the **-rd** option, to turn off remote debugging. If you leave remote debugging enabled, the server will always wait for the specified time-out before continuing with the application.

Time-outs or other failures are logged to a file named **rd.log**, which is created in the DBLDIR directory (or the TEMP directory on Windows Vista/2008 and higher) when the first entry in the file is logged. This file contains the process ID of the instance of the runtime that logged entries, the date and time entries were logged, and specific messages. If you are having a problem debugging remotely, check this file first.



(Windows Vista/2008 and higher) If the TEMP logical is not set in the **Synrc** node in the Windows registry, **rsynd** will put the log file in a system-defined location, most likely somewhere in the C:\Users path. We recommend that you use the Synergy Configuration Program to explicitly set TEMP in the registry for remote debugging. (Remember that **rsynd** only reads environment variables set in the registry, not the environment.)

Deploying Your Distributed Application

This section tells you what needs to be installed and configured on the xfServerPlus machine and the client machine. You will need to refer to this section when setting up your environment for development and testing of your client application, as well as when creating the installation program for your distributed application.

Deploying the Server

1. Install xfServerPlus.
2. Copy your ELBs to the server machine. If you used logicals in the SMC to specify the location of the ELBs, ensure that those logicals are defined in the **xfpl.ini** file (SERVER_INIT.COM on OpenVMS). See [“Defining Logicals” on page 1-4](#).
3. If your **xfpl.ini** file is not in the default location (DBLDIR), set XFPL_INIPATH. See [“Setting the XFPL_INIPATH Environment Variable” on page 3-19](#).
4. Copy the SMC files (**cdt.is?** and **cmpdt.is?**) to the server machine or use the import methods option in the MDU to import the necessary interfaces. For information on moving ISAM files between operating systems, see [“Moving Database Files to Other Systems”](#) in the “Synergy DBMS” chapter of *Synergy Language Tools*. For information on importing methods, see [“Importing and Exporting Methods” on page 2-39](#).

5. If your SMC files are not in the default location (DBLDIR), set XFPL_SMCPATH. See [“Setting the XFPL_SMCPATH Environment Variable for xfServerPlus”](#) on page 2-46.
6. Start xfServerPlus. See [“Running xfServerPlus”](#) on page 3-3 for instructions on running xfServerPlus on your operating system.

Deploying the Client

For information on deploying the client portion of your distributed application, refer to the section for your xfNetLink client:

- ▶ Synergy, see [step 1 on page 5-1](#)
- ▶ Java, see [page 10-1](#)
- ▶ COM, see [page 14-1](#)
- ▶ .NET, see [page 17-1](#)

Configuring xfServerPlus for Remote Data Access

You can access data remotely from xfServerPlus using xfServer. For example, xfServerPlus might be running on one machine, and you want to access data located on another machine, on which xfServer resides. In this scenario, xfServerPlus acts as a client to xfServer. The instructions vary depending on the xfServer security mode and the systems that xfServerPlus and xfServer are running on. For details on xfServer security modes, refer to the section for your operating system in the [“Configuring xfServer”](#) chapter of the *Installation Configuration Guide*.



If xfServerPlus is on OpenVMS, you cannot connect to a remote xfServer process for data access because an OpenVMS system cannot be an xfServer client.

Remote Data Access When xfServerPlus Is on Windows

When xfServerPlus is on Windows, you can access data remotely through xfServer running on Windows, UNIX, or OpenVMS. xfServer may be running in secure, non-secure, restricted (on Windows), or trusted (on UNIX) mode.

- To connect to an xfServer that's running in SECURE mode



If both xfServerPlus and xfServer are on Windows, and xfServer is running in secure mode, and RUSER is not set for the xfServerPlus service, Windows authentication will be attempted. If the xfServerPlus account is a domain account, authentication should succeed. If the xfServerPlus account is a local account, you will need to set RUSER as described in the steps below. See ["Understanding xfServer security"](#) in the "Configuring xfServer" chapter of the *Installation Configuration Guide* for details on xfServer security modes on Windows.

1. On the xfServer machine, create an account using the xfServerPlus account user name and password. (Note that while technically this account does not have to be the same as the xfServerPlus account, we recommend it for the sake of simplicity.)
2. On the xfServerPlus machine, at a command prompt run **setruser** with the **-n** option. When prompted, enter the user name and password you defined on the xfServer machine in [step 1](#).

Setruser -n returns the user name and encoded password to the screen; it does *not* update the RUSER setting in the registry. Leave this output displayed on your screen; you can copy it to create the environment setting. (For more information about **setruser**, see ["The setruser Utility"](#) in the "Configuring xfServer" chapter of the *Installation Configuration Guide*.)

3. Start the Synergy Configuration Program and go to the xfServer/xfServerPlus tab.
4. Select the xfServerPlus service that will be the client to xfServer, click the Modify Service button, and then click the Environment Settings button.
5. Click the Add button that is grouped with the "Settings for *service name*" list.

6. Enter RUSER for the variable name, and then for the value copy and paste the string returned by `setruiser -n` in [step 2](#). Click OK in the Add Environment Setting dialog box. This sets RUSER in the registry in `HKEY_LOCAL_MACHINE\Software\Synergex\Synergy xfServer \serviceName\Synrc`.
7. Click OK in the Environment Settings dialog box and the xfServerPlus Information dialog box, and then click Apply in the Synergy Configuration Program. The service will be stopped and restarted.

► To connect to an *xfServer* that's running in NON-SECURE mode with a default user account

When *xfServer* is running in non-secure mode with a default user account, no additional setup is required on either the *xfServer* or the *xfServerPlus* machine for remote data access. (You do not need to set RUSER on the *xfServerPlus* machine; if RUSER is set, it is ignored.)

► To connect to an *xfServer* that's running in NON-SECURE mode

If *xfServer* is running in non-secure mode without a default user account on Windows, no additional setup is required on either the *xfServer* or the *xfServerPlus* machine for remote data access. (You do not need to set RUSER on the *xfServerPlus* machine; if RUSER is set, it is ignored.)

If *xfServer* is running in non-secure mode without a default user account on UNIX or OpenVMS, do the following:

1. On the *xfServer* machine, create an account using the *xfServerPlus* account user name. (Note that while technically this account does not have to be the same as the *xfServerPlus* account, we recommend it for the sake of simplicity.)
2. On the *xfServerPlus* machine, set RUSER to the user name (a password is not required) that you specified in [step 1](#). Follow [steps 3](#) through [7](#) on [page 3-46](#). In [step 6](#), rather than pasting the string, just enter the user name for the variable value.

Configuring and Running xfServerPlus

Configuring xfServerPlus for Remote Data Access

► To connect to an xfServer that's running in RESTRICTED mode

Restricted mode uses Windows authentication to secure the connection between client and server. Consequently, this configuration is valid only when both xfServerPlus and xfServer are running on Windows and belong to the same domain or trusted domain. In addition, the account used to run xfServerPlus sessions must be a domain account—not a local account. (See the note on [page 3-3](#) for information on using a domain account as the xfServerPlus account.) No additional setup is required. (If RUSER is set, it is ignored.)

► To connect to an xfServer that's running in TRUSTED mode

On the xfServer machine, create an account using the xfServerPlus account user name. This account *must* have the same user name as the xfServerPlus account. Do *not* set RUSER on the xfServerPlus machine.

Remote Data Access When xfServerPlus Is on UNIX

When xfServerPlus is on UNIX, you can access data remotely through xfServer running on Windows, UNIX, or OpenVMS. xfServer may be running in secure, non-secure, or trusted mode (on UNIX). You cannot access data on Windows when xfServer is running in restricted mode, because restricted mode supports only Windows clients.

► To connect to an xfServer that's running in SECURE mode

1. On the xfServer machine, create an account using the xfServerPlus account user name and password. (Note that while technically this account does not have to be the same as the xfServerPlus account, we recommend it for the sake of simplicity.)
2. On the xfServerPlus machine, set the RUSER environment variable to the user name and password you defined on the xfServer machine in [step 1](#).

You should set RUSER in the environment prior to starting xfServerPlus. You can do this in one step with this command:

```
RUSER=`setruser` ;export RUSER
```

This command runs **setruser** (which prompts you for the user name and password), assigns the output to RUSER, and exports it. (For more information about **setruser**, see “[The setruser Utility](#)” in the “Configuring xfServer” chapter of the *Installation Configuration Guide*.)

- To connect to an *xfServer* that's running in NON-SECURE mode with a default user account

When *xfServer* is running in non-secure mode with a default user account, no additional setup is required on either the *xfServer* or the *xfServerPlus* machine for remote data access. (You do not need to set RUSER on the *xfServerPlus* machine; if RUSER is set, it is ignored.)

- To connect to an *xfServer* that's running in NON-SECURE mode without a default user account

When *xfServer* is running in non-secure mode without a default user account on Windows, no additional setup is required on either the *xfServer* or the *xfServerPlus* machine for remote data access. (You do not need to set RUSER on the *xfServerPlus* machine; if RUSER is set, it is ignored.)

When *xfServer* is running in non-secure mode without a default user account *on* UNIX or OpenVMS, do the following:

1. On the *xfServer* machine, create an account using the *xfServerPlus* account user name. (Note that while technically this account does not have to be the same as the *xfServerPlus* account, we recommend it for the sake of simplicity.)
2. On the *xfServerPlus* machine, set RUSER to the user name (a password is not required) you specified in [step 1](#). You don't need to run **setruser** because non-secure mode does not require a password; just set the RUSER environment variable to the user name.

- To connect to an *xfServer* that's running in TRUSTED mode

On the *xfServer* machine, create an account using the *xfServerPlus* account user name. This account *must* have the same user name as the *xfServerPlus* account. Do *not* set RUSER on the *xfServerPlus* machine.

Part II: *xf*NetLink Synergy Edition

***xf*NetLink Synergy: Overview**

*xf*NetLink Synergy Edition provides distributed access to Synergy logic and Synergy data (via access to Synergy logic) from within Synergy applications. *xf*NetLink Synergy is a set of routines distributed with Synergy/DE Professional Series that can be used to call routines that are executed on a remote machine running *xf*ServerPlus.

Remote access to routines enables you to create multi-tier distributed applications by deploying business logic and database access on centralized servers and user interface logic on client machines. Using *xf*ServerPlus and *xf*NetLink Synergy, you can launch full-scale multi-tier applications or incrementally improve the performance of existing applications.

System Overview

Figure 4-1 shows the primary components of a distributed Synergy application using *xf*ServerPlus and *xf*NetLink Synergy. The diagram describes two logical machines:

- A Synergy client running a Synergy application that includes the *xf*NetLink Synergy API
- A Synergy server running *xf*ServerPlus, which handles the remote execution of Synergy routines. The routines are made available for remote execution by including them in an ELB or shared image and defining them in the Synergy Method Catalog (SMC).

Many Synergy applications progress through a similar sequence of steps: initialization (opening files, checking security, loading start-up information, etc.), transaction validation and processing, and termination. Using *xf*ServerPlus and *xf*NetLink Synergy, applications can continue to operate in this fashion.

xfNetLink Synergy: Overview

System Overview

xfNetLink Synergy enables you to use your existing Synergy code without rewriting it, provided that the code is already written in the form of an external subroutine or function. If the routine requires input from or sends messages to the user, or if it might generate untrapped errors, it must be adjusted to work as server-level logic.

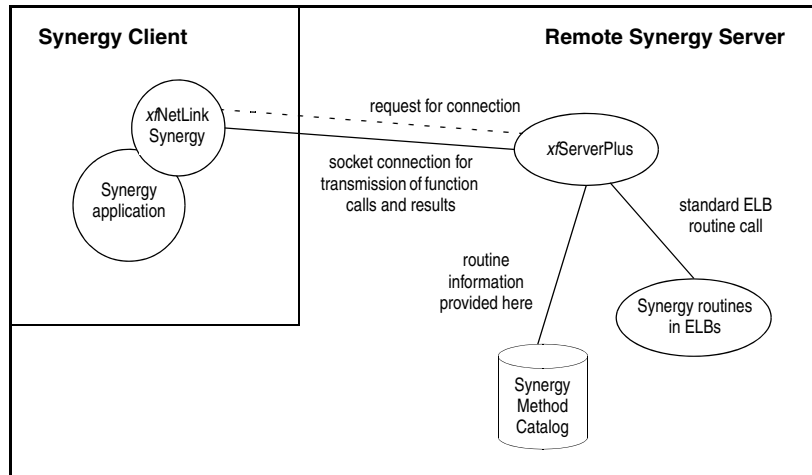


Figure 4-1. xfServerPlus with xfNetLink Synergy.

The client

From the user's perspective, a distributed Synergy application that uses xfNetLink Synergy and xfServerPlus looks no different than any other Synergy application. The Synergy client application contains the user interface and uses the xfNetLink Synergy routines to call centralized server logic. A single client session may create multiple sessions on one server machine.

The server

The Synergy server machine must have at least one xfServerPlus license. The server also requires the following:

- ▶ xfServerPlus must be running.
- ▶ There must be an ELB or shared image containing the Synergy routines you wish to call remotely.
- ▶ There must be a Synergy Method Catalog that identifies the remote routines.

Routines can be executed remotely on more than one server machine; each machine requires an xfServerPlus license.

Creating a connection

To access Synergy logic remotely with a Synergy client, you must establish a connection between the client and the server using `%RX_START_REMOTE`. This routine handles the creation of a socket connection between the Synergy client and a dedicated *xfServerPlus* session running on the server. The use of a dedicated session means that context is maintained from one remote routine call to the next until a shutdown signal is received.

When you call `%RX_START_REMOTE`, you can provide the IP address and port number on which *xfServerPlus* is listening, as well as specify time-out values. If the call is successful and the connection established, the routine returns a network connection ID, which identifies the particular connection.

Calling a Synergy routine remotely

Once the connection has been established, your remote routines are available to the client application. Calling a Synergy routine remotely is similar to calling one locally. You pass the network connection ID, a method ID, and the necessary arguments to the `%RXSUBR` routine.

In the Synergy Method Catalog, the method ID is mapped to a Synergy routine and the ELB or shared image in which it resides. The routine call is received by *xfServerPlus*, which creates the call to the specified Synergy routine. After the Synergy call returns, the results are passed back to the client, and *xfServerPlus* waits for another call.

Configuring & Testing *xf*NetLink Synergy

This chapter gives an overview of the tasks you must perform to set up and use *xf*NetLink Synergy Edition, and explains how to configure *xf*NetLink Synergy and run the *xf*NetLink Synergy test program.

The Big Picture

To successfully set up and run your *xf*NetLink Synergy–*xf*ServerPlus system, you need to do complete the following steps.

1. Install the necessary Synergy software.
 - ▶ Install *xf*ServerPlus on the Synergy server machine. For detailed steps on setting up *xf*ServerPlus, see [“The Big Picture” on page 3-1](#).
 - ▶ Install *xf*NetLink Synergy on the client machine. For development purposes, you’ll need to install Professional Series Development Environment. However, because the *xf*NetLink Synergy routines are part of the runtime, when it comes time to deploy your application, the client machine will need only Core Components (Windows) or the Language (UNIX and OpenVMS).
2. Modularize your existing Synergy code for the routines that you want to call remotely. Encapsulate the code in ELBs or shared images (on OpenVMS). See [chapter 1](#) for more information.
3. Create a user account on the *xf*ServerPlus machine to run *xf*ServerPlus sessions. Give this account privileges that are consistent with your policy for accessing Synergy routines remotely. See [“Running *xf*ServerPlus” on page 3-3](#).
4. Populate the Synergy Method Catalog (SMC) with information about your Synergy routines. You can do this by attributing your code or by entering data with the Method Definition Utility. If you choose to put the SMC somewhere other than DBLDIR, set the XFPL_SMCPATH environment variable. See [chapter 2](#) for details.

Configuring & Testing xfNetLink Synergy

Configuring xfNetLink Synergy

5. In the **xfpl.ini** file, set logging options for the *xfServerPlus* log and set logicals that point to the ELBs you specified in the SMC (if necessary). You may also need to set other options in the **xfpl.ini** file; see [Appendix A](#) for a complete list of **xfpl.ini** configuration settings.

If you choose to put the **xfpl.ini** file somewhere other than DBLDIR, set the XFPL_INIPATH environment variable.

See [chapter 3](#) for information on the log and XFPL_INIPATH; see “[Defining Logicals](#)” on [page 1-4](#) for information on setting logicals that point to your ELBs.

6. (optional) Configure *xfNetLink Synergy* by setting defaults for host name and port, time-out values, and debug options. On Windows, do this in the **synergy.ini** file; on UNIX and OpenVMS, use environment variables. See “[Configuring xfNetLink Synergy](#)” on [page 5-2](#).
7. Create the user interface for your Synergy client. Include code that requests a remote execution session using the *xfNetLink Synergy* API. See [chapter 6](#) for “how-to” information about using the API and [chapter 7](#) for the API routines.
8. Start *xfServerPlus* (**rsynd**) with remote execution enabled. See “[Running xfServerPlus](#)” on [page 3-3](#).
9. Run your client program.

Configuring xfNetLink Synergy

On Windows, *xfNetLink Synergy* is configured with settings in the **synergy.ini** file. On UNIX and OpenVMS, *xfNetLink Synergy* is configured by setting environment variables. By configuring *xfNetLink Synergy* you can

- ▶ specify the host name and port number as defaults.
- ▶ specify time-out values.
- ▶ specify a filename for logging the sent and received packets (for debugging purposes).

The table below shows the variables that can be set. Each is discussed in more detail in the following pages.



See [Appendix A](#) for complete listings of all configuration settings for *xfServerPlus* and *xfNetLink*.

Variables for Configuring xfNetLink Synergy	
Variable	Description
XF_REMOTE_HOST	Machine where <i>xfServerPlus</i> is running
XF_REMOTE_PORT	Port that <i>xfServerPlus</i> is running on
XF_RMTCONN_TIMEOUT	Session time-out for running in regular mode
XF_RMT_DBG_TIMEOUT	Session time-out for running in debug mode
XF_RMT_TIMEOUT	Call time-out for regular session or debug
XFNLS_LOGFILE	Name of file to log packets in

Specifying the Host Name and Port Number

Specifying the host name and port enables *xfNetLink* to read the name and port as defaults. You can also pass this information when you use the `%RX_START_REMOTE` or `%RX_DEBUG_INIT` functions.

► To specify the host name and port as defaults

Specify the host name of the machine on which the *xfServerPlus* service is running and the port number on which it is listening. You don't have to specify both the host name and port; you can specify only one or the other if desired. The port number must be an integer. Neither variable can be null.

On Windows, put this information in the **synergy.ini** file. On UNIX and OpenVMS, specify the host and port as environment variables. For example:

```
XF_REMOTE_HOST=elmo  
XF_REMOTE_PORT=2440
```

Specifying Time-out Values

You can specify two types of time-outs for *xfNetLink Synergy*:

- Connect session for both normal and debug use
- Call (session communication)

You can specify these values in the **synergy.ini** file on Windows and as environment variables on UNIX and OpenVMS.



For xfNetLink Synergy, the request for session time-out ('A' in [figure 5-1](#)) is set for two minutes and cannot be changed. This time-out measures how long xfNetLink will wait to receive an acknowledgment from the connection monitor in xfServerPlus. The connection monitor is responsible for accepting session requests from xfNetLink and signaling the logic server to start a session.

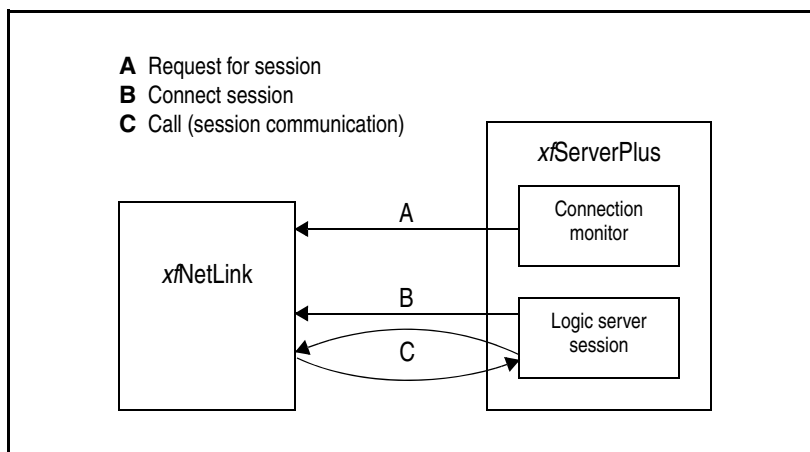


Figure 5-1. xfNetLink time-outs.

Connect session time-out

Once the initial socket communication is established ('A' in [figure 5-1](#)), the connection monitor signals the logic server to start a session. The connect session time-out ('B' in [figure 5-1](#)) measures the time that xfNetLink waits for an acknowledgment from the session started by the logic server. It is measured from the time that xfNetLink receives the acknowledgment from the connection monitor to the time it receives an acknowledgment from the logic server.

This time-out is set separately for normal and debug operation. The default for normal operation is 2 minutes; for debug it is 10 minutes. You will probably want to set the debug time-out to a greater value than the normal time-out since you need to move from one machine to another when starting a debug session (see the note on [page 6-6](#)).

► To specify a connect session time-out

Specify a value in seconds for XF_RMTCONN_TIMEOUT and/or XF_RMT_DBG_TIMEOUT. For example, to specify a 3-minute time-out for normal operation and a 6-minute time-out for debug operation:

```
XF_RMTCONN_TIMEOUT=180  
XF_RMT_DBG_TIMEOUT=360
```

These values can also be set with the *sess_timeout* argument when making a call with %RX_START_REMOTE or %RX_DEBUG_START. A time-out value passed in the routine call overrides the time-out value set as an environment variable or in **synergy.ini**.

Call time-out

The call time-out ('C' in [figure 5-1](#)) measures the length of time that the remote session request should wait for the results of a remote routine call. It is measured for *each* send–receive request between *xf*NetLink and *xf*ServerPlus. The default is 1800 seconds (30 minutes).

► To specify a call time-out

Specify a value in seconds for XF_RMT_TIMEOUT. For example, to specify a 15-minute time-out:

```
XF_RMT_TIMEOUT=900
```

This value can also be set with the *call_timeout* argument when making a call with %RX_START_REMOTE or %RX_DEBUG_START. A time-out value passed in the routine call overrides a time-out set as an environment variable or in **synergy.ini**. The call time-out value can also be overridden at runtime with the routine %RX_RMT_TIMEOUT (see [page 7-15](#)); this affects only the current session.

You can also set a call time-out value for *xf*ServerPlus. See [SET_XFPL_TIMEOUT](#) on [page 1-31](#).

Specifying Debug Options

When you use the %RX_DEBUG_INIT and %RX_DEBUG_START functions to debug the code in your ELBs, you can specify that the packets be written to a file instead of displayed on the screen.

To receive debug trace output, you must also set the *trace_flag* argument in %RX_DEBUG_START to '1'.



This feature is not available on OpenVMS. However, OpenVMS users can view packets on the server side in the *xfServerPlus* log. See [“Using Server-Side Logging” on page 3-29](#).

► To specify a filename for debug trace information

Specify a filename in the **synergy.ini** file (on Windows) or as an environment variable (on UNIX). For example:

```
XFNLS_LOGFILE=Myfile.txt
```

Myfile.txt is created if it does not exist; if the file already exists, additional material is appended to the end. To place the file in a specific location, specify the full path. If you do not specify a path, the file is created in the current working directory. You can also pass a filename when making the call.

Testing xfNetLink Synergy

The **xfnlstst.dbr** test program (**xfnlstst.exe** on OpenVMS), distributed with *xfNetLink Synergy*, can help you determine if your system is set up and working properly.

The **xfnlstst** program runs several tests that send different types of data back and forth between the client and server and make routine calls. This program makes calls to a test ELB or shared image named **xfpl_tst**, which is distributed with *xfServerPlus*. There are entries in the SMC for use by the test program. (These are the methods in the *xfTest* interface in the distributed SMC.) If the ELB or any of the methods are missing, the tests will fail.



If these methods are not present in your SMC, you can import them from the **defaultsmc.xml** file. See [“Importing and Exporting Methods” on page 2-39](#).

► To run the `xfnlstst` program

1. Make sure *xfServerPlus* has been started on the server machine.
2. On Windows and UNIX, at the command line of the machine on which you installed *xfNetLink Synergy*, enter

```
dbr DBLDIR:xfnlstst xfServerMachineName xfServerPort
```

On OpenVMS, you must define **xfnlstst** as a foreign command and then execute it. On the machine on which you installed *xfNetLink Synergy*, enter

```
$ XFNLSTST:==$DBLDIR:XFNLSTST  
$ XFNLSTST xfServerMachineName xfServerPort
```

As the tests run, information is printed to the screen and saved to the **xfnlstst.log** file. You will see a line describing each test and a message stating whether it completed successfully or unsuccessfully. This file is created in the directory from which you ran the test.

If any tests were unsuccessful, check the **xfnlstst.log** file for more information. You may want to also run the *xfServerPlus* test program, **xfsplstst**. It can help you determine whether the problem is on the *xfNetLink* side or the *xfServerPlus* side. See [“Testing xfServerPlus” on page 3-15](#) for more information.

If you cannot solve the problem, call Synergy/DE Developer Support. Be sure to save the **xfnlstst.log** file; your Developer Support engineer needs the information in this file to help you.

Calling Synergy Routines Remotely from Synergy

The *xf*NetLink Synergy Edition Application Program Interface (API) enables you to execute routines remotely. This chapter explains how to use the API routines to make calls from a Synergy client to a remote Synergy server, and offers suggestions on handling errors and troubleshooting problems.

Making Remote Calls

The primary method of calling a remote routine is by using %RXSUBR and the related RX_xxx routines. This approach enables you to make remote routine calls in a way that is very much like using XSUBR.

You can also make remote calls by directly manipulating a routine call block (RCB) with the RCB_xxx routines. See [“Making Remote Calls Using a Routine Call Block” on page 6-3](#).

Making Remote Calls with %RXSUBR

For a complete code sample, see [Appendix C](#). For details on the RX_xxx routines, see [chapter 7](#).

1. .INCLUDE these two files in your program:
 - DBLDIR:rxerr.def
 - DBLDIR:rxapi.def
2. Use %RX_START_REMOTE to initiate a request for a remote session. You can pass the host IP and port in the call, or specify them in the **synergy.ini** file (or as environment variables on UNIX and OpenVMS). (See [“Configuring *xf*NetLink Synergy” on page 5-2](#).) For example:

```
status = %rx_start_remote(netid, ip, port)
```

Calling Synergy Routines Remotely from Synergy

Making Remote Calls

If the call is successful, the function returns the status code `RX_SUCCESS`; otherwise, it returns a status code indicating why the call failed. If the call is successful, `%RX_START_REMOTE` also returns a network connection ID (net ID), which you'll need to make the remote call. The net ID is a handle to an internal memory structure that describes the network connection and its current state.

3. Use `%RXSUBR` to call the remote routine, which then executes in a specified remote session. Pass the network connection ID returned by `%RX_START_REMOTE`, the method ID of the routine you want to call (as defined in the SMC), and up to 253 arguments:

```
xcall rxsubr(netid, "methodid", arg1, arg2)
```

`%RXSUBR` supports passing integer, decimal, implied-decimal, and alpha data types. Packed data is supported on the client side. Arrays of up to nine dimensions are also supported. `%RXSUBR` supports the usual Synergy conventions for returning results from function calls.



You cannot pass `^VAL` and `^REF` arguments to `%RXSUBR`. If a remote routine expects to receive a `^VAL` or `^REF` argument, the arguments must be defined as such in the Synergy Method Catalog. Pass the arguments normally (i.e., by descriptor) to `%RXSUBR`; `xfServerPlus` will convert them to the correct type based on how they are defined in the SMC.

If the call is successful, arguments and results will be updated. If the call fails, a Synergy runtime error will be signaled, and you can retrieve error information. (See [“Handling Errors” on page 6-3.](#))

4. Close the session with `RX_SHUTDOWN_REMOTE`:

```
xcall rx_shutdown_remote(netid)
```



Failure to close the session with `RX_SHUTDOWN_REMOTE` wastes system resources.

Making Remote Calls Using a Routine Call Block

Although %RXSUBR is the primary means of making remote calls, you can also make remote calls by directly manipulating a routine call block (RCB) with the RCB_xxx routines. There are several reasons for using the RCB_xxx routines:

- ▶ You need to pass a variable-length parameter.
- ▶ You need to pass a parameter that exceeds 64K in size. For array parameters, the total size of the array may exceed 64K as long as each individual array element is less than 64K. The maximum size for an array field in Synergy Language is 256 MB.
- ▶ You want to optimize your code. If you're making a call multiple times and you wish to avoid the overhead of building the argument block multiple times, you can optimize your code by using the RCB routines.

For more information, see [“Handling Variable-Length and Large Data” on page 1-12](#) and [“Passing Arrays Larger Than 64K” on page 1-17](#). For instructions on using the RCB routines and the RCB routine syntax, see the [“Synergy Routine Call Block API”](#) chapter of the *Synergy Language Reference Manual*.

Handling Errors

Because applications built using *xfNetLink* Synergy and *xfServerPlus* include multiple programs distributed across multiple machines, error handling design must take into account the possibility of errors occurring in any one of the programs.

Error conditions can occur on either the *xfNetLink* or *xfServerPlus* side of a remote execution session. In a distributed environment, error handling requires multiple layers of processing: the error may occur on a machine that the user does not have access to.

To help you deal with this situation, errors detected by *xfServerPlus* are logged to the application event log (Windows), syslog (UNIX), or operator console (OpenVMS). Additional information can be logged to the *xfServerPlus* log. (See [“Using Server-Side Logging” on page 3-29](#).) This enables system administrators to track problems that have occurred even when they are not reported by a user. In addition, information about errors generated by *xfServerPlus* is stored in memory on the client for each active network connection ID.

Calling Synergy Routines Remotely from Synergy

Handling Errors

There are two categories of *xyServerPlus* errors.

- ▶ Non-fatal errors that *xyServerPlus* can trap and for which it can produce a meaningful diagnostic
- ▶ Fatal errors that *xyServerPlus* cannot trap and which cause abnormal termination of the remote execution session

In both cases, *xyServerPlus* sends a message to the client. When this message is received, relevant information about the error is stored with the associated network connection ID, and a runtime error is signaled on the client (see the table on [page 7-26](#)).

In the event of a non-fatal error, the *xyServerPlus* session remains available for future calls. Detailed error information can be retrieved by calling `RX_GET_ERRINFO` (see [page 7-7](#)).

In the event of a fatal error, the session is aborted. Traceback information can be retrieved by calling `RX_GET_HALTINFO` (see [page 7-9](#)).

You may also see *xyServerPlus* status codes returned to the client. Refer to the table on [page 3-16](#).

Troubleshooting Techniques

Error messages may not always provide enough information to diagnose a problem. When such is the case, you can take advantage of the additional debugging options provided with *xfNetLink* and *xfServerPlus*. You may also want to run the test programs; see “[Testing xfNetLink Synergy](#)” on page 5-6 and “[Testing xfServerPlus](#)” on page 3-15.

Running an *xfServerPlus* Session in Debug Mode

During normal operation, *xfServerPlus* runs as a background process without support for console operations, complex user interfaces, or debugging. This improves efficiency and minimizes memory requirements. However, there may be times when you need to run the debugger on Synergy code in the ELBs that are being called from *xfServerPlus*. By manually connecting an *xfServerPlus* session to your Synergy client application, you can run your Synergy server routines in debug mode so that you can uncover problems that are showing up as errors in your distributed application.

When running in debug mode, you have the option of logging the packets sent from and received by the client. You can display them on screen or write them to a file. See “[Viewing packets on the client side](#)” on page 6-9.



If you do not need to view packets on the client side and the operating system of your *xfServerPlus* is Windows or UNIX, we recommend that you use the Telnet method for debugging. See “[Debugging Remote Synergy Routines via Telnet](#)” on page 3-42 for instructions.

Running in debug mode on Windows and UNIX

Use this procedure if the operating system of your *xfServerPlus* machine is Windows or UNIX.

If your SMC files or **xfpl.ini** file are not in the default location (DBLDIR), you will need to either move them to DBLDIR or set XFPL_SMCPATH and XFPL_INIPATH *in the environment* to point to the location of the files before starting **xfpl.dbr** in [step 4](#). (Note: When XFPL_SMCPATH and XFPL_INIPATH are set in the registry or **synrc**, they are read by **rsynd**. Since **rsynd** is bypassed when you run in debug mode, the registry/**synrc** settings do not get read.)

Calling Synergy Routines Remotely from Synergy Troubleshooting Techniques

1. Use the `%RX_DEBUG_INIT` function to initiate a debug session. This routine binds a port number and IP address for listening, and then returns the port, IP, and a network connection ID. You need to include code that displays the IP address (in hex) and port on the screen. For example:

```
status = %rx_debug_init(netid, ip, port)
writes(output_chan, "IP address is " + %hex(ip))
writes(output_chan, "Port number is " + %string(port))
```

2. When the IP and port display on the screen, write them down. You'll need them in [step 4](#). For example:

```
IP address is 6F16212C
Port number is 1082
```



Once the IP address, etc. displays on the screen, you have a limited amount of time in which to manually start *xfServerPlus* in debug mode on the server machine, specify a breakpoint, and type "go". This time is controlled by the variable `XF_RMT_DBG_TIMEOUT` (see ["Connect session time-out" on page 5-4](#)) or by passing a session time-out value when you make the call. If no time-out is specified, the default value of 10 minutes applies. If you delay longer than this, *xfNetLink* will time out while waiting for a response from *xfServerPlus*.

3. Use `%RX_DEBUG_START` to complete the connection process. Be sure to pass the same net ID that was returned by the corresponding `%RX_DEBUG_INIT` routine. For example:

```
status = %rx_debug_start(netid)
```

At this point, the client application has opened a socket and is waiting for the server to call it back.

4. Go to the machine running *xfServerPlus*, start `xfpl.dbr`, and pass the IP and port to *xfServerPlus*. Type the alpha characters in the IP address in uppercase.

```
dbr -d xfpl hexadecimal_IP port
```

For example:

```
dbr -d xfpl 6F16212C 1082
```

xfServerPlus starts up in the Synergy debugger.

5. Set an initial breakpoint in the `xfpl` program at the `XFPL_DEBUG_BREAK` routine. In the debugger enter

```
break xfpl_debug_break
```

and then enter

```
go
```

`xfServerPlus` is now connected to the client on the specified port. The server waits while the client program resumes and makes its first call. The program will then break at the `XFPL_DEBUG_BREAK` routine. This breakpoint occurs just after `xfServerPlus` has opened the ELB for the first method called by your application. (Note that any ELBs linked to this ELB will also be opened.) The ELB must be opened before you can set breakpoints in the routines within it.

6. If the Synergy routine you need to debug is in one of the opened ELBs, just specify a breakpoint in that routine. If the routine you want to debug is in a different (unopened) ELB, use the `OPENELB` debugger command to open that ELB. (You can also continue running your client application until the ELB is opened by `xfServerPlus`. However, because you set a breakpoint at `XFPL_DEBUG_BREAK`, the program will break at each method call, so using the `OPENELB` command is more efficient.)



For general information about the Synergy debugger, see the “[Debugging Your Synergy Programs](#)” chapter in *Synergy Language Tools*. For details on the `OPENELB` command, see [OPENELB](#) in that same chapter.

Running in debug mode on OpenVMS

Use this procedure if the operating system of your `xfServerPlus` machine is OpenVMS.

1. Make sure `xfServerPlus` is running on an unused port. If necessary, restart it to ensure that it's using a new, unused port.
2. On the machine running `xfServerPlus`, enter

```
$ run DBLDIR:xfpld
```

Calling Synergy Routines Remotely from Synergy Troubleshooting Techniques

You'll see output similar to the following:

```
*****
***  DEBUG 9.3.1    ***
BREAK AT 152 IN XFPL (LAUNCHER.DBL;6) ON ENTRY
%DBG-E-Could not open source file "LAUNCHER.DBL;1"
Dbldb>
*****
```



If you have created shared image logicals for the shared images used by *xfServerPlus*, you can skip [step 3](#). Instead, set a breakpoint for your shared image and routine as described in [step 6](#). You'll then be prompted for the port number ([step 4](#)). Once you start your client program ([step 5](#)), the debug session will break at the breakpoint you set.

3. Set an initial breakpoint in the *xfpl* program at the `XFPL_DEBUG_BREAK` routine. In the debugger enter
`set break xfpl_debug_break`
and then enter
`go`
4. When prompted, enter the port number that *xfServerPlus* is running on (from [step 1](#)).
5. Start your client application in the usual manner (i.e., execute `%RX_START_REMOTE`). After it connects, the debug session will break at the `XFPL_DEBUG_BREAK` routine.
6. Set a breakpoint for your Synergy shared image and routine:

```
set break image/routine
```

(For details, see [BREAK](#) in the “Debugging Your Synergy Programs” chapter of *Synergy Language Tools*.)

Note that if you set a breakpoint at `XFPL_DEBUG_BREAK`, the debugger will break at `XFPL_DEBUG_BREAK` for each method call your client makes.



Although you do not need to use the `OPENELB` debugger command before setting the first breakpoint in your shared image, you may need to use it if your code does an `XSUBR` or `RCB_SETFNC` without specifying a shared image. For details on the `OPENELB` command, see [OPENELB](#) in the “Debugging Your Synergy Programs” chapter in *Synergy Language Tools*.

Viewing Packets

At times, it may be helpful to see the complete packets sent back and forth between the client application and `xfServerPlus`. You can view packets on either the client side or the server side.



What’s the difference between the packets viewed on the client side and those viewed on the server side? If everything is working properly—nothing. The debug trace option (client side) allows you to view packets as they look when sent from and received by `xfNetLink Synergy`. The `xfServerPlus` log (server side) allows you to view packets as they look when received by and sent from `xfServerPlus`. Under normal circumstances they should be the same. Having these two ways to view the information is primarily for your convenience in troubleshooting.

Viewing packets on the client side

The debug trace option allows you to view the actual packets that are sent and received on the client side. This option is available only when running in debug mode (see “[Running an xfServerPlus Session in Debug Mode](#)” on page 6-5).

If encryption is enabled, the log will display a string of 10 asterisks instead of the actual data for packets that contain encrypted data. In addition, the type of encryption (master or slave) will be included in the connection information.



You cannot view packets on the client if you have an OpenVMS server. However, OpenVMS users can view packets on the server side in the `xfServerPlus` log. See “[Using Server-Side Logging](#)” on page 3-29.

Calling Synergy Routines Remotely from Synergy

Troubleshooting Techniques

To activate the debug trace option, set the *trace_flag* argument to '1' when you call the %RX_DEBUG_START routine. By default, the packets are printed to the screen of the client machine. To write them to a file, pass a filename in the %RX_DEBUG_START call or specify the XFNLs_LOGFILE variable in the **synergy.ini** file (or as an environment variable on UNIX).

As shown in the sample below, the connection information prints once for each session; sent and received data prints for each call.

```
-----
Local host: tiger
XFPL connection port: 1217
Call timeout      = 1800 seconds
Session timeout = 600 seconds
-----
send: size = 166
Dxfpl_tst2;10;ID13#1234567890.12;ID10#12345.6789;ID10#12345.6789
;ID11#12345678.91;DE10#1234567890;ID8#0.123456;ID6#1.2345;DE8#12
345678;ID8#123456.7;DE9#123456789;
recv: size = 107
Rxfpl_tst2;005;001ID14#1234567891.12;002ID7#333.334;006ID9#0.998
8332;008DE9#-88991010;009ID8#654321.0;
send: size = 9
S;0;;
```

Viewing packets on the server side

The *xyServerPlus* log shows you the packets that are received by and sent from *xyServerPlus*. To record packets in the *xyServerPlus* log, set the XFPL_DEBUG option to ON in the **xfpl.ini** file. See [XFPL_DEBUG on page 3-36](#) for details on what information is recorded in the log.

***xf*NetLink Synergy API**

This chapter is a reference guide to the functions and subroutines in the *xf*NetLink Synergy Edition API.

`%RX_CONTINUE` 7-2
`%RX_DEBUG_INIT` 7-4
`%RX_DEBUG_START` 7-5
`RX_GET_ERRINFO` 7-7
`RX_GET_HALTINFO` 7-9
`%RX_RMT_ENDIAN` 7-11
`%RX_RMT_INTSIZE` 7-12
`%RX_RMT_OS` 7-13
`%RX_RMT_SYSINFO` 7-14
`%RX_RMT_TIMEOUT` 7-15
`RX_SETRMTFNC` 7-16
`RX_SHUTDOWN_REMOTE` 7-17
`%RX_START_REMOTE` 7-18
`%RXSUBR` 7-23

%RX_CONTINUE

```
result = %RX_CONTINUE(netid[, arg_return_flag])
```

or

```
xcall RX_CONTINUE(netid[, arg_return_flag])
```

%RX_CONTINUE allows a remote routine to continue processing after an %RXSUBR call has timed out.

Return value `result`

The function call return value. If the remote Synergy routine called by %RXSUBR is a function that returns a ^VAL value, the value is returned as the ^VAL result of %RX_CONTINUE. (n)

Arguments `netid`

Network connection ID corresponding to the value you passed with %RXSUBR. (n)

`arg_return_flag`

(optional) Indicates whether you want argument return values from the %RXSUBR call to be updated. If not passed, return values are updated. (n)

Pass zero (0) if the %RXSUBR call passed arguments that return a value (arguments defined as “out” or “in/out” in the SMC) and you want them to be updated when the call completes. This means the call will be completed as though %RXSUBR had not timed out.

Pass a non-zero value if the %RXSUBR call passed arguments that return a value and you do *not* want the return values updated when the call completes. In addition, error packets will not be returned. This means the call will be completed, but the results (including errors) will be thrown away.

Discussion An %RXSUBR call will time out if processing does not complete within a specified length of time. The default call time-out is 30 minutes, but you may set a different value with the XF_RMT_TIMEOUT environment variable or with %RX_RMT_TIMEOUT or when you make the %RX_START_REMOTE call.

When an %RXSUBR call times out, it will signal an \$ERR_TIMEOUT error. You can choose either to call RX_SHUTDOWN_REMOTE and end the session or call %RX_CONTINUE and continue processing. %RX_CONTINUE restarts the call time-out clock, which enables the remote routine to continue processing

on the server for the length of time defined by the call time-out. When %RX_CONTINUE times out, it also signals \$ERR_TIMEOUT. If desired, you can call %RX_CONTINUE multiple times in succession until the call completes. Each call to %RX_CONTINUE restarts the call time-out clock.

Use the same form (function or subroutine) for the %RX_CONTINUE call as you did for the %RXSUBR call that has timed out.

Passing a non-zero value in the *arg_return_flag* argument enables the return packet to be received but not processed, so that you can make another %RXSUBR call and maintain the session context. In other words, the call completes but you throw away the results.

If you call %RX_CONTINUE when an %RXSUBR call has *not* timed out, %RXSUBR will return the error “No current call in progress” (\$ERR_XFNOCALL).

When an %RXSUBR call times out, you cannot make a second %RXSUBR call until a return packet for the first call has been received. At this point you should call %RX_CONTINUE or shut down the session. If, instead, you attempt to make a second %RXSUBR call, %RXSUBR will return the error “Remote call already in progress” (\$ERR_XFINCALL).

Examples In the example below, if the RXSUBR call to method123 times out, RX_CONTINUE is called, resetting the call time-out value and allowing the call to method123 to continue processing.

```
onerror ($ERR_TIMEOUT) timeout
xcall rxsubr(netid, "method123", arg1, arg2)
.
.
.
timeout,
  offerror
  xcall rx_continue(netid)
```

%RX_DEBUG_INIT

```
status = %RX_DEBUG_INIT(netid, listen_ip, listen_port)
```

This function starts a connection to *xfServerPlus* so that you can manually connect an *xfServerPlus* session in debug mode. Connecting in debug mode enables you to debug the code in your ELBs on the server. This function is used only for Windows and UNIX servers—not for OpenVMS.

Return value *status*

Indicates whether the connection was successful. (n)

If the connection is made successfully, the status returned is RX_SUCCESS.

If the connection is not made successfully, a Synergy socket error is generated.

Arguments *netid*

Network connection ID describing the connection to the machine on which remote routines are executed. This value is set by %RX_DEBUG_INIT. (n)

listen_ip

The IP address of the machine the client is running on. (a or n)

listen_port

The port of the machine the client is running on. (n)

Discussion %RX_DEBUG_INIT binds a port and IP for listening, and then returns the port number and IP address in the corresponding fields. You have to display those values on the screen in a way that is appropriate for your operating system. Use %RX_DEBUG_START to complete the debug connection process.

See “[Running in debug mode on Windows and UNIX](#)” on page 6-5 for more information.

%RX_DEBUG_START

```
status = %RX_DEBUG_START(netid, [trace_flag], [trace_file],
&                                [call_timeout][, sess_timeout])
```

This function completes the process of connecting in debug mode that was started by %RX_DEBUG_INIT. After the listening port and IP have been displayed, use this function to complete the connection so that you can start a debug session. This function is used only for Windows and UNIX servers—not for OpenVMS.

Return value *status*

Indicates whether the request for a remote session was successful. (n)

The status codes returned are the same as for %RX_START_REMOTE. See the table on [page 7-20](#).

Arguments *netid*

Network connection ID describing the connection to the machine on which remote routines are executed. You should pass the net ID returned by the corresponding %RX_DEBUG_INIT routine. (n)

If you call %RX_DEBUG_START and pass a net ID that was not returned by %RX_DEBUG_INIT, the runtime error \$ERR_XFNOINIT is signaled.

If the net ID is invalid, \$ERR_NOTNETHND or \$ERR_INVNETHND is signaled. See the table on [page 7-26](#) for explanations of these codes.

trace_flag

(optional) Indicator that client packets should be logged. The default is false. (n)

trace_file

(optional) The file the packets should be logged in. (a)

If a filename is passed, it is used. Otherwise, **synergy.ini** (or the corresponding environment variable on UNIX) is checked for the value XFNL_LOGFILE. If no file is found, the packets are displayed on the screen of the client machine.

call_timeout

(optional) The length of time (in seconds) that the remote session request should wait for the results of a remote routine call. The default is 1800 seconds (30 minutes). (n)

This time-out is measured for each send–receive request between *xfNetLink* and *xfServerPlus*. (This is ‘C’ in [figure 5-1 on page 5-4](#).)

If a call time-out is passed, it is used. Otherwise, **synergy.ini** (or the corresponding environment variable on UNIX) is checked for the value **XF_RMT_TIMEOUT**. If no time-out is specified there or if the value is ≤ 0 , the default is used. This value is associated with the network connection ID and may be changed at runtime with **%RX_RMT_TIMEOUT** (see [page 7-15](#)).



For more information on setting time-outs for *xfNetLink*, see [“Specifying Time-out Values” on page 5-3](#).

sess_timeout

(optional) The length of time (in seconds) that the remote session request should wait for a session connection from *xfServerPlus*. The default is 600 seconds (10 minutes). (n)

This time-out is measured from the time that *xfNetLink* receives the connection request acknowledgment from the connection monitor to the time it receives an acknowledgment from the session started by the logic server. (This is ‘B’ in [figure 5-1 on page 5-4](#).) Note that the time-out for the request for session from the connection monitor is 2 minutes (‘A’ in [figure 5-1](#)): if no acknowledgment is received from the connection monitor within 2 minutes, a time-out will occur no matter how *sess_timeout* is set.

If a session time-out is passed, it is used. Otherwise, **synergy.ini** (or the corresponding environment variable on UNIX) is checked for the value **XF_RMT_DBG_TIMEOUT**. If no time-out is specified there or if the value is ≤ 0 , the default time-out is used.

Discussion See [“Running in debug mode on Windows and UNIX” on page 6-5](#) for more information.

RX_GET_ERRINFO

```
xcall RX_GET_ERRINFO(netid, remote_err_descriptor)
```

This subroutine returns information about non-fatal errors generated by *xfServerPlus*. You can call this routine to get more information about a non-fatal runtime error generated during an %RXSUBR call. See the record format below for the information returned.

Arguments

netid

Network connection ID corresponding to the value set by %RX_START_REMOTE. (n)

remote_err_descriptor

A record containing the fields in the remote execution error packet. The record format is

```
group rx_stderror
  method_id           ,a32
  num_of_errors       ,i4
  error_num           ,i4
  description         ,a128
  clarifying_desc     ,a256
endgroup
```

Discussion In the event of a non-fatal error, *xfServerPlus* sends a message to the client. When this message is received, information about the error is stored with the associated network connection ID and a runtime error is signaled on the client. The *xfServerPlus* session remains available for future calls.

The record structure shown above is defined in **rxerr.def**, located in DBLDIR. To extract error information, you have to .INCLUDE the **rxerr.def** file.

Examples To .INCLUDE **rxerr.def**:

```
.define RX_ERR_DEF
record errinfoec
.include "DBLDIR:rxerr.def"
.undefine RX_ERR_DEF
```

xfNetLink Synergy API

RX_GET_ERRINFO

Error handling code:

```
onerror ($ERR_XFREQPARM) handle_parmerr, handle_other
    ;trap parm err separately, and then trap others
    xcall rxsubr(netid, "methodid", arg1, arg2, arg3)
offerror
.
.
.
handle_parmerr,
    xcall RX_GET_ERRINFO(netid, errinfo)
    xcall handle_parmerr(errinfo)
    xreturn

handle_other,
    xcall RX_GET_ERRINFO(netid, errinfo)
    case %ERROR of
        begincase
            .
            .
            .
        endcase
    else
        .
        .
        .
    xreturn
```

RX_GET_HALTINFO

```
xcall RX_GET_HALTINFO(netid, halt_err_descriptor)
```

This subroutine returns information about fatal errors generated by *xfServerPlus*. You can call this routine to get more information about a fatal runtime error generated during an %RXSUBR call. See the record format below for information returned.

Arguments

netid

Network connection ID corresponding to the value set by %RX_START_REMOTE. (n)

halt_err_descriptor

A record containing the fields in the remote fatal packet. The record format is

```
group rx_fatalerror
  subroutine_name      ,a32
  error_line_number    ,i4
  error_num            ,i4
  sys_error_number     ,i4
  prog_name            ,a128
  error_text           ,a128
endgroup
```

Discussion In the event of a fatal error, *xfServerPlus* sends a message to the client. When this message is received, information about the error is stored with the associated network connection ID, a runtime error is signaled on the client, and the session is aborted.

The record structure shown above is defined in **rxerr.def**, located in DBLDIR. To extract error information, you have to **.INCLUDE** the **rxerr.def** file.

Examples

```
To .INCLUDE rxerr.def:

.define RX_FATAL_DEF
record haltinfoec
.include "DBLDIR:rxerr.def"
.undefine RX_FATAL_DEF
```

xfNetLink Synergy API

RX_GET_HALTINFO

Error handling code:

```
onerror ($ERR_XFHALT) handle_fatal      ;trap fatal errors
    xcall rxsubr(netid, "methodid", arg1, arg2, arg3)
offerror
.
.
.
handle_fatal,
    xcall RX_GET_HALTINFO(netid, haltinfoec)
    xcall mylogfunc(haltinfoec)
xreturn
```

%RX_RMT_ENDIAN

```
status = %RX_RMT_ENDIAN(netid, endian_type)
```

This function returns the integer endian code for the platform that *xfServerPlus* is running on for a particular network connection ID.

Additional operating system information can be obtained with %RX_RMT_INTSIZE, %RX_RMT_OS, and %RX_RMT_SYSINFO.

Return value *status*

Indicates whether the query was successful. Status returned is RX_SUCCESS. (n)

Arguments *netid*

Network connection ID corresponding to the value set by %RX_START_REMOTE. (n)

If the net ID is invalid, \$ERR_NOTNETHND (581) or \$ERR_INVNETHND (571) is signaled. See the table on [page 7-26](#) for explanations of these codes.

endian_type

Integer endian type of the platform *xfServerPlus* is running on. These are included in **rxapi.def**. (n)

0 = RX_ENDIAN_LITTLE

1 = RX_ENDIAN_BIG

%RX_RMT_INTSIZE

status = %RX_RMT_INTSIZE(*netid*, *int_size*)

This function returns the integer size for the platform that *xfServerPlus* is running on for a particular network connection ID.

Additional operating system information can be obtained with %RX_RMT_ENDIAN, %RX_RMT_OS, and %RX_RMT_SYSINFO.

Return value *status*

Indicates whether the query was successful. Status returned is RX_SUCCESS. (n)

Arguments *netid*

Network connection ID corresponding to the value set by %RX_START_REMOTE. (n)

If the net ID is invalid, \$ERR_NOTNETHND (581) or \$ERR_INVNETHND (571) is signaled. See the table on [page 7-26](#) for explanations of these codes.

int_size

The integer size of the platform *xfServerPlus* is running on. These are included in **rxapi.def**. (n)

0 = RX_INTSIZE_32

1 = RX_INTSIZE_64

2 = RX_INTSIZE_128

%RX_RMT_OS

```
status = %RX_RMT_OS(netid, os)
```

This function returns the operating system that *xfServerPlus* is running on for a particular network connection ID.

Additional operating system information can be obtained with %RX_RMT_ENDIAN, %RX_RMT_INTSIZE, and %RX_RMT_SYSINFO.

Return value *status*

Indicates whether the query was successful. Status returned is RX_SUCCESS. (n)

Arguments *netid*

Network connection ID corresponding to the value set by %RX_START_REMOTE. (n)

If the net ID is invalid, \$ERR_NOTNETHND (581) or \$ERR_INVNETHND (571) is signaled. See the table on [page 7-26](#) for explanations of these codes.

os

The operating system *xfServerPlus* is running on. These are included in **rxapi.def**. (n)

1 = RX_OS_UNIX

2 = RX_OS_VMS

3 = RX_OS_WIN

%RX_RMT_SYSINFO

status = %RX_RMT_SYSINFO(*netid*, *os*, *endian_type*, *int_size*)

This function returns operating system information (remote *xfServerPlus* operating system, endian type, and integer size) associated with a particular net ID. Individual values can be retrieved with %RX_RMT_OS, %RX_RMT_ENDIAN, and %RX_RMT_INTSIZE.

Return value *status*

Indicates whether the query was successful. Status returned is RX_SUCCESS. (n)

Arguments *netid*

Network connection ID corresponding to the value set by %RX_START_REMOTE. (n)

If the net ID is invalid, \$ERR_NOTNETHND (581) or \$ERR_INVNETHND (571) is signaled. See the table on [page 7-26](#) for explanations of these codes.

os

The operating system *xfServerPlus* is running on. These are included in **rxapi.def**. (n)

1 = RX_OS_UNIX
2 = RX_OS_VMS
3 = RX_OS_WIN

endian_type

Integer endian type for the platform *xfServerPlus* is running on. These are included in **rxapi.def**. (n)

0 = RX_ENDIAN_LITTLE
1 = RX_ENDIAN_BIG

int_size

The integer size for the platform *xfServerPlus* is running on. These are included in **rxapi.def**. (n)

0 = RX_INTSIZE_32
1 = RX_INTSIZE_64
2 = RX_INTSIZE_128

%RX_RMT_TIMEOUT

status = %RX_RMT_TIMEOUT(*netid*, *result_timeout*)

This function sets the call time-out value for the specified network connection ID.

Return value *status*

Indicates whether the query was successful. (n)

Status codes returned are RX_SUCCESS and RX_BADTIMSPEC. See the table on [page 7-20](#) for explanations of these codes.

Arguments *netid*

Network connection ID corresponding to the value set by %RX_START_REMOTE. (n)

If the net ID is invalid, \$ERR_NOTNETHND (581) or \$ERR_INVNETHND (571) is signaled. See the table on [page 7-26](#) for explanations of these codes.

result_timeout

The time (in seconds) that the Synergy client should wait for the results of a remote routine call. (n)

Discussion There is a call time-out value associated with each network connection ID. This value determines how long the Synergy client will wait for the result of a remote Synergy routine call. It is measured for each send–receive request. (This is ‘C’ in [figure 5-1 on page 5-4](#).)

The call time-out is set at start-up with %RX_START_REMOTE, but can be changed at runtime. If it is changed, the new value remains in effect for all subsequent routine calls until it is changed again. If the time-out value is invalid (i.e., ≤ 0), it is not changed. The call time-out can also be set in the **synergy.ini** file or as an environment variable (see “[Specifying Time-out Values](#)” on [page 5-3](#)).

RX_SETRMTFNC

```
xcall RX_SETRMTFNC(rcbid, netid, method_id[/encrypt])
```

This subroutine enables you to specify the remote routine (by method ID) and the network connection ID so that when a call is made with %RCB_CALL, the remote routine is called.

Arguments

- rcbid**
The identifier for the routine call block, returned by %RCB_CREATE. (n)
- netid**
Network connection ID corresponding to the value returned by %RX_START_REMOTE. (n)
- method_id**
The unique identifier (in the Synergy Method Catalog) of the routine being called. (a)
- /encrypt**
(optional) Encryption is desired for this method.

Discussion If you created a dynamically-generated routine call block (RCB) and wish to call it remotely, use this subroutine to set the method ID and network connection ID in the RCB.

The **/encrypt** switch is required when slave encryption is enabled on the *xfServerPlus* machine and the method is marked for encryption in the SMC. The **/encrypt** switch is optional when slave encryption is enabled on the *xfServerPlus* machine and the method is not marked for encryption in the SMC. The **/encrypt** switch is not required when master encryption is enabled. See [“Using Encryption” on page 3-23](#) for more information on encryption.

See the [“Synergy Routine Call Block API”](#) chapter of the *Synergy Language Reference Manual* for the RCB routine syntax and instructions on using the RCB_xxx routines.

RX_SHUTDOWN_REMOTE

```
xcall RX_SHUTDOWN_REMOTE(netid)
```

This subroutine sends a request to close the dedicated *xfServerPlus* session, which causes the running Synergy process to terminate and releases the memory associated with the network connection ID.



Failure to close the session with RX_SHUTDOWN_REMOTE wastes system resources.

Arguments *netid*

Network connection ID for the connection that you want to shut down. (n)

%RX_START_REMOTE

```
status = %RX_START_REMOTE(netid, [ip], [port], [call_timeout]  
&                                [, sess_timeout])
```

This function initiates a request for an *xfServerPlus* session. It handles the creation of a socket connection between the Synergy client and a dedicated *xfServerPlus* session running on the server. **%RX_START_REMOTE** returns a status code that indicates the success or failure of the call. If the call is successful, a network connection ID is returned in the *netid* argument.

Return value status

A status code indicating the result of the request for a remote session. Status codes are defined in the file **DBLDIR:rxapi.def**, which you should **.INCLUDE** in programs that use *xfNetLink Synergy*. See the table on [page 7-20](#) for explanations of the status codes and what you can do to resolve the problems they represent. (n)

If **%RX_START_REMOTE** returns **RX_SUCCESS**, execution is ready to proceed.

Arguments netid

Network connection ID. The net ID is a handle to a memory structure containing information that describes the location and state of the *xfServerPlus* session that was created. (n)

This value is set by **%RX_START_REMOTE** and should be passed to all subsequent *xfNetLink* calls directed at the *xfServerPlus* session that it represents. If **%RX_START_REMOTE** does not return **RX_SUCCESS**, *netid* is set to 0.

ip

(optional) Host name or IP address of the machine *xfServerPlus* is running on. (a or n)

If an IP is passed, it is used. Otherwise, **synergy.ini** (or the corresponding environment variable on UNIX and OpenVMS) is checked for the value **XF_REMOTE_HOST**. If no IP is specified there, the connection errors out.

port

(optional) The port number *xfServerPlus* is running on. The default is 2356. (n)

If *port* is passed, it is used. Otherwise, **synergy.ini** (or the corresponding environment variable on UNIX and OpenVMS) is checked for the value XF_REMOTE_PORT. If no port is specified there, the default is used.

`call_timeout`

(optional) The length of time (in seconds) that the remote session request should wait for the results of a remote routine call. The default is 1800 seconds (30 minutes). (n)

This time-out is measured for each send–receive request between *xfNetLink* and *xfServerPlus*. (This is ‘C’ in [figure 5-1 on page 5-4](#).)

If a call time-out is passed, it is used. Otherwise, **synergy.ini** (or the corresponding environment variable on UNIX and OpenVMS) is checked for the value XF_RMT_TIMEOUT. If no time-out is specified there or if the value is ≤ 0 , the default time-out is used. This value is associated with the network connection ID and may be changed at runtime with %RX_RMT_TIMEOUT (see [page 7-15](#)).



For more information on setting time-outs for *xfNetLink*, see [“Specifying Time-out Values” on page 5-3](#).

`sess_timeout`

(optional) The length of time (in seconds) that the remote session request should wait for a session connection from *xfServerPlus*. The default is 120 seconds (2 minutes). (n)

This time is measured from the time that *xfNetLink* receives the connection request acknowledgment from the connection monitor to the time it receives an acknowledgment from the session started by the logic server. (This is ‘B’ in [figure 5-1 on page 5-4](#).) Note that the time-out for the request for session from the connection monitor is 2 minutes (‘A’ in [figure 5-1](#)): if no acknowledgment is received from the connection monitor within 2 minutes of making the call, a time-out will occur no matter how *sess_timeout* is set.

If a session time-out is passed, it is used. Otherwise, **synergy.ini** (or the corresponding environment variable on UNIX and OpenVMS) is checked for the value XF_RMTCONN_TIMEOUT. If no time-out is specified there or if the value is ≤ 0 , the default time-out is used.

Examples See [Appendix C](#).

xfNetLink Synergy API
%RX_START_REMOTE

%RX_START_REMOTE Status Codes			
Number	Status code	Meaning	What to do
0	RX_SUCCESS	Session started.	N/A
12301	RX_INVLOGIN	Remote session log-in failed.	Make sure the user name and password you specified when registering the <i>xfServerPlus</i> service are valid.
12302	RX_NOTLICENSED	<i>xfServerPlus</i> not licensed.	Call your Synergy/DE customer service representative to request an XFPL license.
12303	RX_LICENSELMT	Too many consecutive <i>xfServerPlus</i> sessions running.	Increase your connection licenses or make sure your application disconnects in an appropriate fashion when it no longer needs the remote session.
12304	RX_LICENSEEXP	Extended demo period expired.	Call your Synergy/DE customer service representative for new licenses.
12305	RX_RSYNDTIMEOUT	Client timed out waiting to start remote session.	Adjust the session time-out argument in the %RX_START_REMOTE call or change the value for the session time-out in the synergy.ini file (or change the environment variable on UNIX and OpenVMS). If the problem persists, check with your system administrator.
12306	RX_NOSTART	OS level problem in spawning or forking the remote session.	Verify that the <i>xfServerPlus</i> account has read/write permission to the DBLDIR directory. Check with your system administrator.
12307	RX_INVRESULT	Unexpected error returned by <i>xfServerPlus</i> .	Call Synergy/DE Developer Support.
12308	RX_NODBLDIR	The DBLDIR environment variable is not set for the registered service.	See “Setting the XFPL_SMCPATH Environment Variable for <i>xfServerPlus</i>” on page 2-46 . Follow the instructions for setting XFPL_SMCPATH for a <i>specific instance</i> of <i>xfServerPlus</i> , but substitute DBLDIR for XFPL_SMCPATH.
12309	RX_NOXFSP	<i>xfServerPlus</i> not running on the port specified.	Verify that <i>xfServerPlus</i> is running on the port specified with XF_REMOTE_HOST.

%RX_START_REMOTE Status Codes (continued)			
Number	Status code	Meaning	What to do
12310	RX_XFSPLINCOMPAT	xfNetLink client version incompatible with server version.	Upgrade either your client or your server to the higher version. Running a newer client with an older server is not a supported configuration.
12311	RX_NOMEM	Insufficient memory on server during start-up.	Upgrade memory or decrease number of running processes.
12312	RX_OUTRNG	Internal "out of range" error in server start-up.	Call Synergy/DE Developer Support.
12313	RX_INVHDL	Internal "invalid memory handle" on server start-up.	Call Synergy/DE Developer Support.
12314	RX_NOHOSTSPEC	No host specified.	Add host name or IP address of remote server to your start-up call or specify it in synergy.ini (or as an environment variable on UNIX and OpenVMS).
12315	RX_BADPORTNO	Port number not numeric.	Change the port argument or check that a valid port was specified in synergy.ini (or as an environment variable on UNIX and OpenVMS).
12316	RX_BADTIMSPEC	Time not numeric.	Correct time-out argument(s) to pass number of seconds.
12317	RX_BADLOCALHOST	Cannot get local host name (machine where xfNetLink Synergy is running).	Check with your system administrator.
12318	RX_BADHOSTNAME	Cannot get host name (machine where xfServerPlus is running).	Check with your system administrator.
12319	RX_NOUSRBATPRV	No batch privilege for user name.	(Windows) The user account used to start xfServerPlus does not have the "log on as a batch job" user right set. This error can occur only if you are using xfServerPlus version 7.1.5 or earlier. Using a version of the server that predates your client version is not a supported configuration. Upgrade xfServerPlus to the current version.

xfNetLink Synergy API
%RX_START_REMOTE

%RX_START_REMOTE Status Codes (continued)			
Number	Status code	Meaning	What to do
12320	RX_NORUNLICENSE	No Runtime license.	Verify that there is a Runtime license on the server machine.
12322	RX_LICENSEACCESS	Cannot access License Manager (synd).	Check with your system administrator.
10000 level errors	N/A	Socket errors.	

%RXSUBR

```
result = %RXSUBR(netid, method_id[/encrypt][, arg, ...])
```

or

```
xcall RXSUBR(netid, method_id[/encrypt], result[, arg, ...])
```

or

```
xcall RXSUBR(netid, method_id[/encrypt][, arg, ...])
```

This routine is the primary means of calling remote Synergy routines, given an existing connection to an *xfServerPlus* session created with %RX_START_REMOTE.

Return value *result*

A function call return value.

- ▶ If the remote Synergy routine is a function that returns a ^VAL value, the value is returned as the ^VAL result of %RXSUBR. (n)
- ▶ If the remote Synergy routine is a function that returns a non-^VAL value, the value must be passed as an extra argument at the beginning of the argument list. (a or n)
- ▶ If the remote Synergy routine is a subroutine, no value is returned.

Arguments *netid*

Network connection ID corresponding to the value set by %RX_START_REMOTE. (n)

method_id

The unique identifier (in the SMC) of the routine being called. (a)



The method ID is case sensitive.

/encrypt

(optional) Encryption is desired for this method.

arg

(optional) The argument list. A routine may be called with no arguments. The maximum number permitted is 253.

Discussion %RXSUBR supports passing integer, decimal, implied-decimal, and alpha data types. Packed data is supported on the client side. Arrays of up to nine dimensions are also supported. %RXSUBR supports the usual Synergy conventions for returning results from function calls.



You cannot pass ^REF and ^VAL arguments to %RXSUBR. If a routine expects to receive a ^VAL or ^REF, the arguments must be defined as such in the SMC (see [“Defining Parameters” on page 2-28](#)). Pass the arguments normally to %RXSUBR; *xfServerPlus* will convert them to the correct type based on how they are defined in the SMC.

- ▶ To call an external function that returns a ^VAL, use
`result = %RXSUBR(netid, method_id [, args])`
- ▶ To call an external function that returns a type other than ^VAL, use
`xcall RXSUBR(netid, method_id, result [, args])`
- ▶ To call a subroutine, use
`xcall RXSUBR(netid, method_id [, args])`

The **/encrypt** switch is required when slave encryption is enabled on the *xfServerPlus* machine and the method is marked for encryption in the SMC. The **/encrypt** switch is optional when slave encryption is enabled on the *xfServerPlus* machine and the method is not marked for encryption in the SMC. The **/encrypt** switch is not required when master encryption is enabled. See [“Using Encryption” on page 3-23](#) for more information on encryption.

When %RXSUBR is called, the following validations take place:

- ▶ The network connection ID must be valid.
- ▶ The method ID must be passed.
- ▶ The data must be of a supported type.
- ▶ The number of arguments cannot exceed 253.
- ▶ If the method is marked for encryption in the SMC, the **/encrypt** switch must be passed.

If any of these items are not validated, a Synergy runtime error is signaled (see the table on [page 7-26](#)) and processing ceases. If all items are validated, %RXSUBR calls the specified Synergy routine.

If the call to the routine is successful, %RXSUBR returns any updated arguments or results as indicated in the Synergy Method Catalog.

If the call is not successful, a Synergy runtime error is signaled (see the table on [page 7-26](#)). To get more detailed information about a particular error, call RX_GET_ERRINFO (see [page 7-7](#)) or RX_GET_HALTINFO (see [page 7-9](#)), as indicated in the table.

Examples ► This example calls a function that returns a ^VAL.

```
status = %rxsubr(netid, "cust_info", name, SSN)
.
.
.
function cust_info      ,^val
    a_name              ,a
    a_ssn               ,n
proc
.
.
.
endfunction
```

► This example calls a function that returns a non-^VAL.

```
xcall rxsubr(netid, "cust_info", status, name, SSN)
.
.
.
function cust_info
    a_name      ,a
    a_ssn       ,n
proc
.
.
.
end
```

► This example calls a subroutine.

```
xcall rxsubr(netid, "cust_info", name, SSN)
.
.
.
subroutine cust_info
    a_name      ,a
    a_ssn       ,n
```

```
proc
.
.
.
end
```

See [Appendix C](#) for additional code samples.

Synergy Runtime Errors Signaled by %RXSUBR			
Number	Runtime error	Caused by	What to do
111	\$ERR_TIMEOUT	Client timed out after waiting specified length of time for call results.	Extend the call time-out (see “Specifying Time-out Values” on page 5-3) or optimize the called routine. If the problem persists, check with your system administrator. You can use %RX_CONTINUE (see page 7-2) to continue processing after a remote call has timed out.
320	\$ERR_NETPROB	Tried to make a call on a disconnected socket or lost socket connection during call.	Try again.
550	\$ERR_XFBADPKTID	Error occurred parsing returned response.	Call Synergy/DE Developer Support. See RX_GET_ERRINFO for more information.
551	\$ERR_XFBADMTHID	Method ID is too long.	Correct the method ID; the limit is 31 characters. See RX_GET_ERRINFO for more information.
552	\$ERR_XFNUMPARMS	Invalid number of arguments sent.	Check routine call against definition in the SMC.
553	\$ERR_XFBADPKT	Error occurred parsing returned response.	Call Synergy/DE Developer Support. See RX_GET_ERRINFO for more information.
554	\$ERR_XFBADTYPE	Argument type didn’t correspond to the definition in the SMC.	Check routine call against definition in the SMC. See RX_GET_ERRINFO for more information.
555	\$ERR_XFREQPARM	Required argument not sent.	Check routine call against definition in the SMC.
556	\$ERR_XFBADARRAY	Error occurred mapping array element.	Call Synergy/DE Developer Support. See RX_GET_ERRINFO for more information.

Synergy Runtime Errors Signaled by %RXSUBR (continued)			
Number	Runtime error	Caused by	What to do
557	\$ERR_XFIOERR	File I/O error occurred on server.	See RX_GET_ERRINFO for more information.
558	\$ERR_XFMETHKNF	Method ID (key) is not found in the SMC.	Check routine call against definition in the SMC. Remember that the method ID is case sensitive. See RX_GET_ERRINFO for more information.
559	\$ERR_XFRTNNF	Method not found in specified ELB or shared image.	Make sure the correct ELB is specified in the SMC and that logicals are set correctly in the xfpl.ini file to point to the ELB (see “ Defining Logicals ” on page 1-4).
560	\$ERR_XFNOCONN	Connection to host was lost.	Restart session. If transaction was midstream, you may need to restore the system to a valid state before restarting.
561	\$ERR_XFHALT	Fatal untrapped error in Synergy routine.	Check routine for untrapped errors. Make corrections to your code. Be sure to check the number and type of parameters. Restore system as required and restart session. See RX_GET_HALTINFO for more information.
563	\$ERR_SRVNOTSUP	Feature not supported on remote server.	Upgrade xfServerPlus to the current version.
564	\$ERR_XFUNKERR	Unknown error reported by xfServerPlus.	The server returned an error that was not recognized by the client. Check the xfpl.log file, which records the error even though the client is unable to receive it. This error usually happens when an older xfNetLink client is communicating with a newer xfServerPlus server. To solve this problem, update your client version. If your versions already match, call Synergy/DE Developer Support.
569	\$ERR_SYNSOCK	Socket error.	See RX_GET_ERRINFO for more information.

Synergy Runtime Errors Signaled by %RXSUBR (continued)			
Number	Runtime error	Caused by	What to do
571	\$ERR_INVNETHND	Network connection ID is not a valid memory handle.	Correct the netid argument.
581	\$ERR_NOTNETHND	Network connection ID is a valid memory handle but not a net ID.	Correct the netid argument.
592	\$ERR_XFINCALL	Remote call already in progress	Use %RX_CONTINUE (see page 7-2) to complete the timed-out %RXSUBR call before making another remote call. Or, shut down the session with RX_SHUTDOWN_REMOTE. See RX_GET_ERRINFO for more information.
596	\$ERR_XFMETHCRYPT	Method requires encryption. The method is marked for encryption in the SMC, but the data was sent unencrypted.	Use the /encrypt switch on the %RXSUBR call or, if this method does not really need to be encrypted, clear the checkbox in the MDU. See %RXSUBR on page 7-23 for information on /encrypt and “Using Encryption” on page 3-23 for information on encryption in general.
597	\$ERR_XFSERVNOSEC	Encryption not enabled on server. The client has sent encrypted data, but encryption is not enabled on the server.	Start rsynd with the -encrypt option or change your client code to remove the /encrypt switch. See %RXSUBR on page 7-23 for information on /encrypt and “Using Encryption” on page 3-23 for information on encryption in general.

Part III: *xf*NetLink Java Edition ▶

xfNetLink Java: Overview

This chapter contains a technical overview of xfNetLink Java Edition and xfServerPlus and tells you what hardware and software you'll need to implement a distributed computing system with xfNetLink Java.

System Requirements

To build a distributed computing system with xfNetLink Java and xfServerPlus, you'll need the items listed below in addition to xfNetLink and xfServerPlus.

- ▶ JDK™ (Java Development Kit) version 1.2 or higher, available free from Sun Microsystems. See the release notes (REL_XFNJ.TXT) for more information.
- ▶ (optional) Java development environment
- ▶ Web server and web browser, if you want to develop a web application
- ▶ (optional) Firewall; *strongly* recommended if your application is running on the Internet

Technical Overview of xfNetLink Java and xfServerPlus

In a multi-tier application developed with xfNetLink Java and xfServerPlus, the user interface is written in Java or generated using JavaScript, while the database and application logic are provided by routines written in Synergy Language. Most often, the user interface is implemented with JavaServer Pages.

Using JavaServer Pages

You can create a web-based user interface for your Synergy applications by generating dynamic HTML on the server side using JavaServer Pages (JSP). The user interface runs on the client's workstation and is downloaded upon demand. Code in the user interface makes calls back to the originating web server for access to centralized data and application logic. A dedicated Java application running on the web server translates Java method calls and Java data into Synergy method calls and Synergy data.

xfNetLink Java: Overview

Technical Overview of xfNetLink Java and xfServerPlus

When you use JSP, all communication with the client is in HTML, avoiding any issues caused by browser differences or firewalls. No processing is done on the client. For more information about JSP, see <http://java.sun.com/products/jsp> in the Java section of the Sun website.

Figure 8-1 shows the primary components of a distributed application using xfNetLink Java, xfServerPlus, and JavaServer Pages. This diagram describes three logical machines:

- ▶ A client workstation running a web browser
- ▶ A web server that serves up JavaServer Pages. The xfNetLink Java classes that handle the creation of a conduit between the end-user and the remote Synergy functionality also reside on the web server. If you are using Java class wrappers, your JAR file must also be on the web server.
- ▶ A Synergy server using xfServerPlus to handle the remote execution of Synergy methods (Synergy functions and subroutines that have been made available for use with xfNetLink)

The Client

From the end-user's perspective, a distributed application using Synergy and Java looks no different than any other web application. Using a web browser, the user connects to a web page on your site that provides access to your application, logs in according to your requirements, and starts using your application. You'll need to write JavaServer Page logic that generates dynamic HTML.

The Web Server

The web server requires the following software and files:

- ▶ The web server software
- ▶ For JavaServer Pages, a JSP servlet container
- ▶ The Java Virtual Machine (JVM™), JDK version 1.2 or higher
- ▶ A Java application or JavaServer Pages that use the xfNetLink Java classes for remote Synergy execution
- ▶ The HTML pages that contain the JavaScript
- ▶ Java class files and JAR file (if using Java class wrappers)

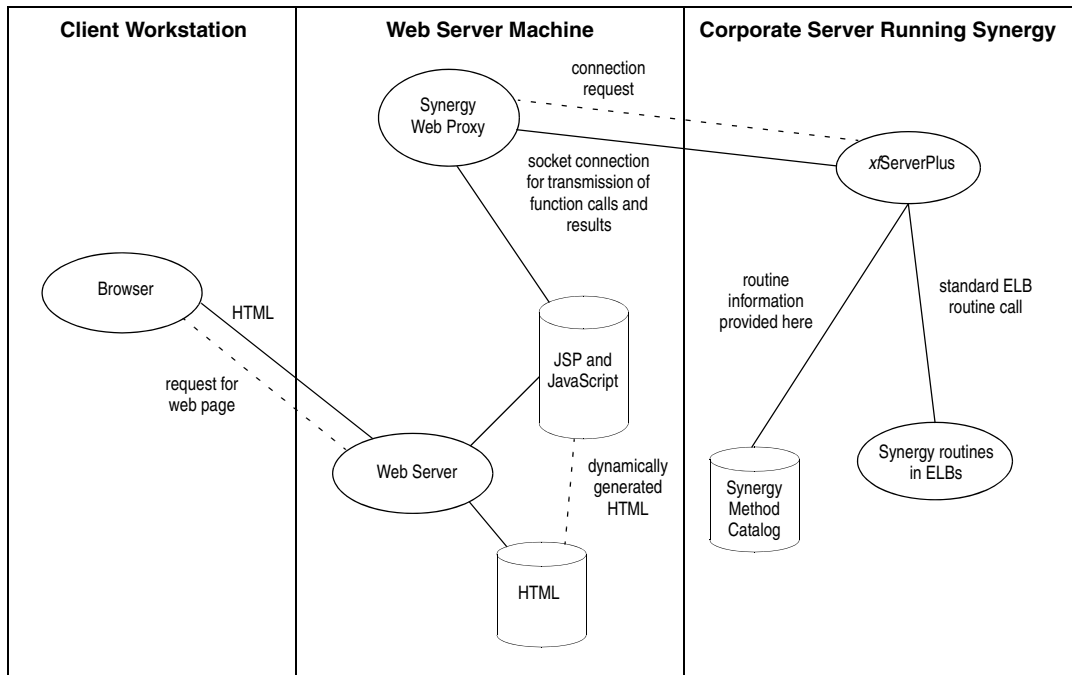


Figure 8-1. Components in an *xfNetLink Java* and *xfServerPlus* implementation using JSP.

The Synergy Host

The Synergy host machine requires the following:

- ▶ *xfServerPlus* must be running.
- ▶ There must be an ELB or shared image containing the Synergy routines that you wish to access over the Web. *xfNetLink Java* enables you to use your existing Synergy code without rewriting it, provided that the code is already written in the form of an external subroutine or function. If the routine requires input from or sends messages to the user, or if it might generate untrapped errors, it must be adjusted to work as server-level logic.
- ▶ The Synergy routines must be identified in the Synergy Method Catalog.

Creating a Connection

The first step in making Synergy logic accessible over the Web is to establish a remote session connection between the client and *xfServerPlus*.

When you initiate a connection, you provide the IP address and port number on which *xfServerPlus* is listening. *xfNetLink* Java then sends a request to *xfServerPlus* for a dedicated session. With the completion of this connection, a dedicated Java to Synergy bridge exists between the client and *xfServerPlus*.



Starting the session requires some network overhead. By ensuring that the user is doing something while the connections are being established, you can decrease any perception of performance problems.

Calling a Synergy Method

Once the *xfNetLink*–*xfServerPlus* connection has been established, your Synergy routines and data are available to the Web. [Figure 8-2](#) shows the message flow for calling a Synergy method from the Java application running on the web server.

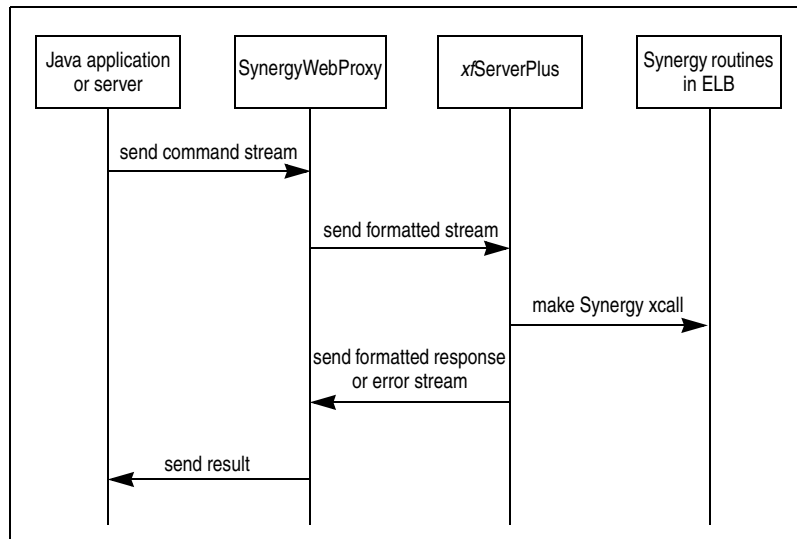


Figure 8-2. Message flow for calling a Synergy method.

The Java call

To call a Synergy method from Java, you pass a method ID and the necessary parameters to the `SynergyWebProxy.xcall()` method. In the Synergy Method Catalog, the method ID is mapped to a Synergy function or subroutine and the ELB in which it resides. If you are using Java class wrappers, the `xcall()` is made behind the scenes (via the `SWPConnect` class) when you call the `connect()` method or when you connect automatically by calling one of your own methods.

When the `xcall()` method is invoked, the method ID and the parameter data are passed to the `SynergyWebProxy`'s data translation methods, where they are formatted into a stream of information. The `SynergyWebProxy`'s network transport methods then send this formatted stream across the socket to `xfServerPlus`.

The `SynergyWebProxy` waits for a response in the form of a result stream or error stream. This stream is read by the network transport methods and passed to the data translation methods, which parse the stream, update the contents of the variables holding the data (or create an exception), and return control to the calling Java class.

How `xfServerPlus` processes the call

The method/routine call stream is received by the `xfServerPlus` network transport routines and then passed to the data translation routines for parsing. The data translation routines create the call to the specified Synergy routine. After the Synergy call returns, the results are passed back to the data translation routines, which create a result or error stream. The stream is then passed back to Java by the network transport routines, and `xfServerPlus` waits for another method/routine call.

The Synergy Method Catalog

At the heart of the data translation process lies the Synergy Method Catalog (SMC). The SMC identifies each method, its subroutine or function name, and the ELB or shared image in which the method can be found. The SMC also provides detailed information on the type and length of the input and output parameters, as well as the type and length of the function results, that are being transmitted across the socket. You can populate the SMC with this information by entering it manually through the Method Definition Utility or by attributing your code, running `dbl2xml` to create an XML file, and then loading that file into the SMC. (See [chapter 2](#) for details on the SMC.)

xfNetLink Java: Overview

Technical Overview of xfNetLink Java and xfServerPlus

If you are using Java class wrappers, methods within the SMC are grouped into interfaces. You then select these interfaces when creating the class wrappers.

The *xfServerPlus* data translation routines use the information in the SMC to allocate adequate memory for the data that is passed to (and possibly updated by) the Synergy Language routines, and to ensure that the information passed across the socket is translated into the correct Synergy Language data types. See [Appendix B](#) for more details on data translation.

Creating Java Class Wrappers

This chapter gives an overview of the tasks you must perform to set up and use *xfNetLink* Java Edition and explains how to generate Java class wrappers and build a JAR file for your Synergy methods. The Java class wrappers feature wraps the calls to *xfNetLink* Java, which enables Java developers to call Synergy logic using a more natural Java interface. The Java class wrappers feature includes support for Java connection pooling, passing structures as parameters, passing collections of structures, and passing binary data such as JPEG files.

You do not have to generate Java class wrappers in order to use *xfNetLink* Java. You can continue to use the *xfNetLink* Java classes directly. The classes are documented in [chapter 12](#).

The Big Picture

This section lists all the steps that need to be completed to successfully create a distributed system using *xfNetLink* Java's class wrappers feature. Note that these steps may not all be done by you, the Synergy developer. For example, you may create the Java JAR file and give it to a Java developer to create a web front-end for your application. The Java developer may also work with a web-page developer to create the HTML portions of the user interface. However, this section should give you a feel for everything that needs to be done, regardless of who does it.

1. Install JDK version 1.2 or higher.
2. Install the necessary Synergy software. The components you need to install vary depending on your set-up (e.g., what OS your source files are on, where you intend to do the development, and so on). Note the following:
 - The Java class wrapper generation tools are part of the *xfNetLink* Java installation. After installing, set your classpath. See [“Setting the Classpath” on page 9-3](#).

Creating Java Class Wrappers

The Big Picture

- ▶ The Professional Series Development Environment (or Workbench) installation includes the SMC/ELB comparison utility, the MDU and SMC files, **genxml**, and the test skeleton generation utility. On UNIX and OpenVMS, these utilities are part of the Synergy Language installation.
 - ▶ *xfServerPlus* must be installed on your Synergy server machine. The *xfServerPlus* (*xfSeries*) installation also includes the MDU, SMC files, and **genxml**. For detailed steps on setting up *xfServerPlus*, see [“The Big Picture” on page 3-1](#).
3. Modularize your existing Synergy code for the routines that you want to access remotely and encapsulate them in ELBs or shared images. See [chapter 1](#), [“Preparing Your Synergy Server Code.”](#)
 4. Populate the Synergy Method Catalog with information about your Synergy routines. You can do this by attributing your code or by entering data with the Method Definition Utility. As you do this, you’ll group the routines into interfaces. If you choose to put the SMC somewhere other than DBLDIR, set the XFPL_SMC_PATH environment variable. See [chapter 2](#), [“Defining Your Synergy Methods.”](#)
 5. In the **xfpl.ini** file, set logging options for the *xfServerPlus* log and set logicals that point to the ELBs you specified in the SMC (if necessary). You may also need to set other options in the **xfpl.ini** file; see [Appendix A](#) for a complete list of **xfpl.ini** configuration settings.

If you choose to put the **xfpl.ini** file somewhere other than DBLDIR, set the XFPL_INIPATH environment variable.

See [chapter 3](#), [“Configuring and Running xfServerPlus,”](#) for information on the log and XFPL_INIPATH. See [“Defining Logicals” on page 1-4](#) for information on setting logicals that point to your ELBs.
 6. Create a user account on the *xfServerPlus* machine to run *xfServerPlus* sessions, and then start *xfServerPlus*. See [“Running xfServerPlus” on page 3-3](#).
 7. If you’re using Workbench, create a Synergy/DE Java Component project and add it to the workspace. Specify the component information: a name and location for the JAR file, the location of the SMC, the location of repository files (if applicable), the package name, the interfaces you want to include, and (if desired) alternate names for those interfaces. See [“Creating a Synergy/DE Java Component Project” on page 9-4](#).

8. If you're using Workbench, use the menu option to generate the Java class wrappers. See [“Generating Java Class Wrappers” on page 9-7](#).

If you're not using Workbench, run the **genxml** utility to create a Synergy XML file, and then run **genjava** to create the Java class wrappers. See [“Creating a Java JAR File from the Command Line” on page 9-8](#).

9. (optional) Modify the generated Java source files if necessary. See [“Editing the Java Source Files” on page 9-16](#).
10. If you're using Workbench, use the menu option to compile the Java classes and create a JAR file. See [“Building the JAR File” on page 9-8](#).

If you're not using Workbench, run the batch file that was created by **genjava** to compile the files and create the JAR file. See [“Building the JAR File” on page 9-13](#).

11. Distribute the JAR file and other necessary files to the Java developer, who will then use the JAR file when writing the client-side code for your distributed application. See [“Deploying the Client” on page 10-1](#) and [“Using Your JAR File” on page 10-11](#).

Setting the Classpath

The *xfNetLink* classes are distributed in the file **xfnljav.jar**. Your classpath must point to this JAR file so the Java compiler can find the *xfNetLink* classes. In addition, your classpath must include the **xercesImpl.jar** and **xml-apis.jar** files (for the XML parser required by the **genjava** utility) and the directory in which you installed *xfNetLink* Java. The classpath setting must be a system-wide setting, not just a user login setting.

For example:

```
classpath=c:\xfNLJava;c:\xfNLJava\xfnljav.jar;  
c:\xfNLJava\xercesImpl.jar;c:\xfNLJava\xml-apis.jar;.
```



When you deploy your application at a customer site, you do not need to include the **xercesImpl.jar** file, **xml-apis.jar** file, and the *xfNLJava* directory in the classpath. They are required only when you are building the class wrappers and JAR file.

Creating a Java JAR File in Workbench

The component generation tools enable you to create a Java JAR file of Synergy methods. The JAR file can be used in any Java application to make remote calls to Synergy routines. To create a JAR file, you must

- ▶ create a Synergy/DE Java Component Project in Workbench and specify information about how the JAR file should be constructed.
- ▶ generate the Java class wrappers.
- ▶ edit the Java code if necessary.
- ▶ build the JAR file.



You can generate class wrappers and build the JAR file from the command line rather than from Workbench. See [“Creating a Java JAR File from the Command Line” on page 9-8.](#)

Creating a Synergy/DE Java Component Project

1. In Workbench, select Project > New, and then select Synergy/DE Java Component from the list of project types. (Expand the Synergy/DE node in the Project type display to select Synergy/DE Java Component.)
2. Give the project a name in the Project name field, and indicate whether you want to create a new workspace or add the project to an existing workspace. For more information on using this dialog, as well as information about basic Workbench features, see the Workbench online help or the [“Developing Your Application in Workbench”](#) chapter of *Getting Started with Synergy/DE*.
3. Specify the following information in the Component Information dialog box. (This dialog displays automatically when you create a new Java component project. If you need to display it later, select Build > Component Information.)

Name. Enter a name for the JAR file. The default is the project name.

Directory. Specify the directory in which to place the JAR file. The default is the project location. If you enter a logical in this field, it must be followed by a colon (e.g., DBLDIR:). This directory will also hold the intermediate files that are used to create the Java class wrappers and JAR file: the XML file, the batch file, and the manifest file. See the explanation for the Package name field below for more information on the directory structure created when you generate class wrappers.

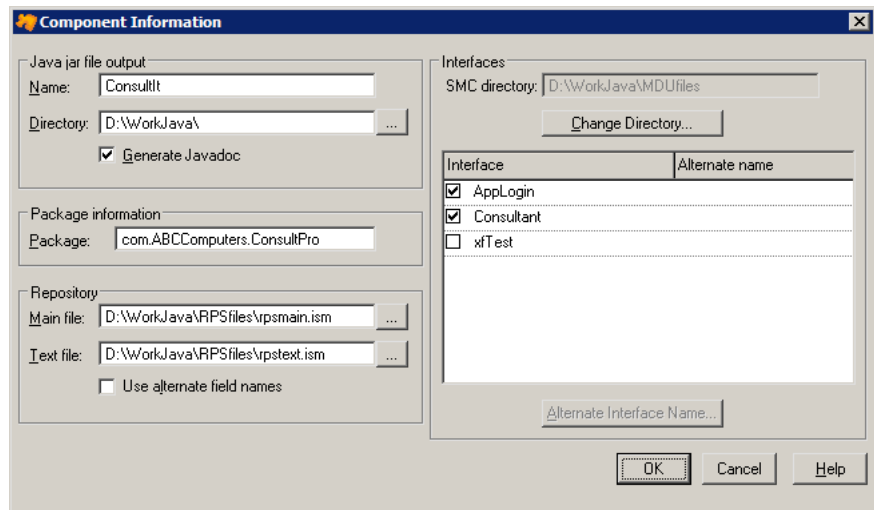


Figure 9-1. The Component Information dialog box for a Synergy/DE Java Component project.

Generate Javadoc. Indicate whether you want to generate Javadoc™ HTML files. Javadoc is generated from information in the .java files and serves as the API documentation for the Java classes you are generating. In order to have useful Javadoc, you must include comments for your methods and parameters. The Javadoc HTML files are created when you build the JAR file. See “[Generating Javadoc](#)” on page 9-16 for additional information.

Package. (optional) Specify a name for the package that will contain the generated classes. The default is the JAR file name. Use dot notation to indicate multiple directory levels. The maximum length for package name is 101 characters.

For example, the package name com.CompanyName.ProductName results in the directory structure com\CompanyName\ProductName, which is created beneath the directory specified in the Directory field. If you specify a multilevel package name, the generated class files are placed in the lowest level directory (ProductName, in our example).

Repository main file. (optional) If any of the methods that will be included in this JAR file pass structures as parameters, specify the location of the repository main file for those structures. This must match the repository that was used when entering data in the SMC. If you specify a main file, you must also specify a text file.

The default is the value of the environment variable `RPSMFIL`. If it is not defined, the default is `RPSDAT:rpsmain.ism`. If neither `RPSMFIL` nor `RPSDAT` are defined, the default is `rpsmain.ism` in the path specified in the Working directory property of the project. If the Working directory is not defined, the default is `rpsmain.ism` in the location where the project is stored.

Repository text file. (optional) If any of the methods that will be included in this JAR file pass structures as parameters, specify the location of the repository text file for those structures. This must match the repository that was used when entering data in the SMC. If you specify a text file, you must also specify a main file.

The default is the value of the environment variable `RPSTFIL`. If it is not defined, the default is `RPSDAT:rpstext.ism`. If neither `RPSTFIL` nor `RPSDAT` are defined, the default is `rpstext.ism` in the path specified in the Working directory property of the project. If the Working directory is not defined, the default is `rpstext.ism` in the location where the project is stored.

Use alternate field names. Indicate whether you want to use the value in the Alternate name field in Repository instead of the value in the Name field as the property name. If not selected, the field name in the structure becomes the property name in the Java class. If selected, the value in the Alternate name field is used if it exists; else, the value in the Name field is used.

SMC directory. This field displays the path for the Synergy Method Catalog that this JAR file uses. If set, Workbench will use `XFPL_SMCPATH` to find the SMC files; else, the default is `DBLDIR`. To change the SMC directory, click the Change Directory button to display the Browse for SMC Directory dialog box. Navigate to the directory, double-click to select it, and click OK. The selected path will display in the SMC directory field and the list of interfaces will be refreshed, displaying all interfaces in the selected SMC.

Interfaces. Select the interfaces you want to include in the JAR file by clicking in the box to the left of the interface name. A Java class, named with the interface name, will be generated for each selected interface.

Alternate name. (optional) You can provide an alternate name for any interface you select. To specify an alternate name, highlight a selected interface, and then click the Alternate Interface Name button and enter another name in the dialog that displays.

You may wish to use an alternate name if the interface name in the SMC is not what you want users to see as the class name. In addition, if your interface names differ only in case, or if you have a structure with the same name as an interface,

you can use the alternate interface name to avoid having numbers appended to the class names. See the note on [page 2-25](#).

To remove an alternate interface name, clear the checkbox for the interface and then reselect it.

4. Click OK in the Component Information dialog box.

Generating Java Class Wrappers

1. To generate the Java class wrappers, select Build > Generate Class Wrappers in Workbench. This command will do the following:
 - ▶ Run the SMC/ELB Comparison utility (**smc_elb.exe**; see [page 2-55](#) for more information).
 - ▶ Create a Java source file for each interface you selected and for each structure and group within the selected interfaces.
 - ▶ Create a manifest file that names the classes in the JAR file and indicates they are JavaBeans™.
 - ▶ Create a batch file to be used later to create the JAR file.
 - ▶ Create files named *filename_srclist.dat* and *filename_clslist.dat* (where *filename* is the name of the JAR file), which are used in conjunction with the batch file to create the JAR file.
 - ▶ Add the **.java** files to the project. You can access them from the Projects tab or the Symbols tab in the project toolbar.

If you have previously generated class wrappers for this project, you will be prompted to overwrite them. If you're regenerating wrappers for the *same interfaces*, the **.java** files will be overwritten and any changes you made to them will be lost.

2. Edit the files as necessary before building the JAR file. See [“Editing the Java Source Files” on page 9-16](#).

Creating Java Class Wrappers

Creating a Java JAR File from the Command Line

Building the JAR File

When you have edited the Java source files, (see [“Editing the Java Source Files” on page 9-16](#)), you are ready to build the JAR file.

1. Verify that the Java component project is the active project.
2. Select Build > Build Jar File. This command will do the following:
 - ▶ Compile the .java files into .class files.
 - ▶ Build the JAR file.
 - ▶ Include the manifest file in the JAR file.
 - ▶ Produce the Javadoc HTML files if the “Generate Javadoc” flag was set in the Component Information dialog box when you created the class wrappers.

Creating a Java JAR File from the Command Line

If your Synergy source files are not on Windows, or if you don’t use Workbench, follow the instructions in this section to create the Java class wrappers, Javadoc, and JAR file from the command line. To create a Java JAR file from the command line, you’ll need to do the following:

1. Use the **genxml** utility to create a Synergy XML file. See [“The genxml Utility” on page 9-9](#).
2. Use the **genjava** utility to create the Java classes. See [“The genjava Utility” on page 9-12](#).
3. Edit the Java source files if necessary. See [“Editing the Java Source Files” on page 9-16](#).
4. Run the batch file to compile the classes, create the JAR file, and (optionally) create the Javadoc HTML files. See [“Building the JAR File” on page 9-13](#).

The genxml Utility

The **genxml** utility creates a Synergy XML file from SMC method definitions and repository structure definitions. This is an intermediate step in creating Java class wrappers.

This utility runs on all supported Synergy/DE platforms. **Genxml** is installed in the DBLDIR directory.



The **genxml** utility checks structure sizes in the SMC against the corresponding structures in the repository and reports a warning if there are discrepancies. Although the XML file is generated anyway, you should use the MDU's Verify Catalog utility to update the structure sizes in the SMC. (See [“Verifying Repository Structure Sizes and Enumerations” on page 2-42.](#)) Failure to do so can cause errors at runtime (because the structure information in the component will differ from that in the SMC).

Syntax `dbr genxml -f xmlFilename -i intName [-a altIntName]
[-d targetDir] [-s smcDir] [-m rpsMain -t rpsText] [-n]
[-v msgLevel] [-?]`

Arguments `-f xmlFilename`

The name to use for the XML file. This name will also be used for the JAR file. You can include the complete path if desired. The extension “.xml” will be appended to this filename if you don't specify an extension.

`-i intName`

Name of the interface from the SMC to include in the XML file. You may pass multiple interface names; each must be preceded with the `-i` option. A Java class will be created for each interface specified. Remember, the interface name is case sensitive.



When using alternate names, sequence matters. The `-a` option must follow the `-i` option that it applies to. You may specify multiple interface names, and each may have an associated alternate name immediately following it.

`-a altIntName`

(optional) Alternate interface name. Use this name for the interface previously specified with the `-i` option. **Genxml** uses the associated `-i` interface to pull methods from the SMC; the alternate name is included in the XML file and is

Creating Java Class Wrappers

Creating a Java JAR File from the Command Line

used as the class name when **genjava** is run. If you pass multiple interface names, each may have an alternate name. Each alternate name must be preceded with the **-a** option. See the examples on [page 9-11](#).

-d *targetDir*

(optional) The target directory for the XML file. If not passed, the XML file is created in the directory specified with the **-f** option. If no directory is specified with **-f**, the file is created in the current directory.

-s *smcDir*

(optional) Directory where the SMC files (**cdt.is?** and **cmpdt.is?**) are located. If not passed, DBLDIR is used.

-m *rpsMain*

(optional) Full path to the repository main file that contains the structures referenced in the SMC. Use with **-t**. This option is used if you are passing structures as parameters. If not passed, **genxml** uses the environment variable RPSMFIL to determine the name of the repository main file; if that is not set, it uses **RPSDAT:rpsmain.ism**. If RPSDAT isn't set, **genxml** looks in the current directory for **rpsmain.ism**.

-t *rpsText*

(optional) Full path to the repository text file that contains the structures referenced in the SMC. Use with **-m**. This option is used if you are passing structures as parameters. If not passed, **genxml** use the environment variable RPSTFIL to determine the name of the repository text file; if that is not set, it uses **RPSDAT:rpstext.ism**. If RPSDAT isn't set, **genxml** looks in the current directory for **rpstext.ism**.

-n

(optional) Indicates that you want to use the value in the Repository Alternate name field instead of the value in the Name field. This option pertains only when you are passing structures as parameters. If not set, the field name in the structure becomes the property name in the class. If passed, the value in the Alternate name field is used if it exists; else, the value in the Name field is used.

-v *msgLevel*

(optional) Level of verbosity in messages:

0 = no messages

1 = error messages and warnings

2 = everything included in level 1, plus success messages (default)

3 = everything included in level 2, plus return codes and the location of the SMC and repository files

-?

(optional) Displays a list of options and the version number for **genxml**.

Examples This example creates a Synergy XML file named **ConsultIt.xml**. This will also be the name of the JAR file. The XML file will include information about two interfaces, AppLogin and Consultant. The target directory for the XML file is c:\work, which is also where the SMC files are located.

```
dbr DBLDIR:genxml -f ConsultIt -i AppLogin -i Consultant  
-d c:\work -s c:\work
```

This example uses alternate interface names. Instead of classes named AppLogin and Consultant, the JAR file will have classes named Login and Consult.

```
dbr DBLDIR:genxml -f ConsultIt -i AppLogin -a Login  
-i Consultant -a Consult -d c:\work -s c:\work
```

On OpenVMS, you'll need to define **genxml** as a foreign command and then execute it. In the example below, we have quoted the XML filename and the interface names to preserve the case.

```
$ GENXML:==$DBLDIR:GENXML  
$ GENXML -F "ConsultIt" -I "AppLogin" -I "Consultant" -  
-D SYS$WORK: -S SYS$WORK:
```

The genjava Utility

The **genjava** utility is a Java program that uses the Synergy XML file created by **genxml** and outputs the following:

- ▶ Java source files (**.java** files). There will be one file for each interface you specified when running **genxml**, as well as a file for each structure and group in those interfaces. The source files are created as a Java package, named with the name you specify with the **-p** option.
- ▶ A manifest file (named with the XML filename), which will be used when creating the JAR file.
- ▶ A batch file (also named with the XML filename), which will be used to compile the classes, create the JAR file, and generate Javadoc.
- ▶ Files named *filename_srclist.dat* and *filename_clslist.dat* (where *filename* is the name of the XML file), which are used in conjunction with the batch file to create the JAR file.

The **genjava** utility is installed in the `xfNLJava` directory.

Syntax `java genjava -f xmlFilename [-p packageName] [-d targetDir] [-j] [-v msgLevel] [-?]`

Arguments `-f xmlFilename`

The full name and path of the XML file generated with **genxml**. If you do not specify the file extension, **“.xml”** will be appended to the end of the filename.

`-p packageName`

(optional) The name of the package that the generated classes will belong to. Use dot notation to indicate multiple levels. The maximum length for package name is 101 characters.

If passed, a directory structure corresponding to the package name will be created within *targetDir* (or the current directory; see below), and the generated class files will be placed in the lowest level directory in the structure. For example, the package name `com.CompanyName.ProductName` would result in the directory structure `com\CompanyName\ProductName`, and the generated classes would be placed in `ProductName`.

If not passed, the XML filename is used as the package name, and so the subdirectory for the generated class files is named with the XML filename.

- d *targetDir*
(optional) Specifies the directory in which the batch file, the manifest file, and, later, the JAR file will be created. If not passed, the files are created in the current working directory.
- j
(optional) Indicates that you want to create Javadoc files. This option adds a command to create Javadoc to the batch file. (See [“Generating Javadoc” on page 9-16.](#))
- v *msgLevel*
(optional) Level of verbosity in messages.
 - 0 = no messages
 - 1 = error messages
 - 2 = error messages and informational messages (default)
- ?
(optional) Displays a list of options for the utility.

Examples The example below creates **.java** files from **ConsultIt.xml**. The package name for the classes is **com.ABCComputers.ConsultPro**, and Javadoc will be created. The **.bat** (**.com** on OpenVMS), **.mf**, and **.dat** files will be created in the **c:\work** directory. The directory structure **com\ABCComputers\ConsultPro** will be created within **c:\work**, and the generated **.java** files will be placed in the **ConsultPro** directory. This is a continuation of our previous example, so the new files will be named **AppLogin.java** and **Consultant.java**.

```
java genjava -f c:\work\ConsultIt.xml  
-p com.ABCComputers.ConsultPro -d c:\work -j
```

Building the JAR File

When you have edited the **.java** files (see [“Editing the Java Source Files” on page 9-16](#)), you are ready to build the JAR file.

The **genjava** utility created a batch file (**.bat** for Windows and UNIX; **.com** for OpenVMS) named with the XML filename and placed it in the directory you specified with the **-d** option. The batch file includes commands to compile the Java classes, create the JAR file, and—if you included the **-j** option when running **genjava**—generate Javadoc HTML files.

Creating Java Class Wrappers

Understanding the Generated Classes

Run the batch file. For example, to continue with our previous scenario:

```
c:\work\ConsultIt
```

The JAR file (named **ConsultIt.jar** in our example) will be placed in the **-d** directory. If you created Javadoc, the HTML files will be placed in the package directory and in a subdirectory within that. See [“Generating Javadoc” on page 9-16](#) and [“Deploying the Client” on page 10-1](#) for more information about these files.

Understanding the Generated Classes

When you generate Java class wrappers from Workbench or with **genjava**, a **.java** file is created for each selected interface in the SMC as well as for each structure and group within the selected interfaces. These classes use the *xfNetLink* Java classes internally to connect with *xfServerPlus* and pass data. Because the Java class wrappers “wrap” this functionality within the implementation of the class, you do not need to access the *xfNetLink* Java classes directly. If you want to know more about the *xfNetLink* Java classes, see [chapter 12](#).

Procedural Classes

The procedural classes (which are derived from the interfaces in the SMC) contain your own user-defined methods as well as a number of utility methods. If you browse a procedural class in Workbench, you’ll see that for each of your methods, the method signature shows the parameter type as specified in the SMC. More information about the parameters is available when you generate Javadoc.

The following public utility methods are included in every procedural class:

<code>connect()</code>	<code>setUserString()</code>
<code>debugInit()</code>	<code>setxfCallTimeout()</code>
<code>debugStart()</code>	<code>setxfHost()</code>
<code>disconnect()</code>	<code>setxfLogfile()</code>
<code>getConnect()</code>	<code>setxfLogging()</code>
<code>getUserString()</code>	<code>setxfMaxport()</code>
<code>getxfHost()</code>	<code>setxfMinport()</code>
<code>getxfLogfile()</code>	<code>setxfPort()</code>
<code>getxfMaxport()</code>	<code>shareConnect()</code>
<code>getxfMinport()</code>	<code>usePool()</code>
<code>getxfPort()</code>	

These methods are used to connect to and disconnect from *xjServerPlus* and to set and get properties, such as the host name and port number. There are also methods for setting a call time-out value, for writing a string to the *xjServerPlus* log, and for specifying client-side logging options. See the method reference on [page 10-35](#) for more information.

Structure Classes

The structure classes are built from your structure definitions in the repository. Fields in the repository structure become properties in the structure class, named with the repository field name.



By default, the properties are named with the repository field names. See [“Passing Structures as Parameters” on page 1-8](#) for details on overriding the default name.

These properties are accessed through “set” and “get” methods. (You cannot access the properties directly; you must use the set and get methods.) The structure class will include a set and a get method for each property. For example, if the repository field is named “cusname”, the property will be “Cusname”, and there will be two methods to access the value of that property, `setCusname()` and `getCusname()`.

If there are groups within your repository structures (or fields declared in the repository as “struct” data type), a class will be created for each group, and each field in the group will be a property of that class. There will be a set and get method for each property.

In addition to the set and get methods, there are several other methods in the structure classes that are used internally to create a representation of a Synergy structure within the Java language. You don’t need to use these methods, but they have to be public so that they can be called from another class.

For more information about the structure and group classes, see [“Using Structures” on page 10-15](#).

Editing the Java Source Files

You may need to edit the generated Java source files to add methods, such as validation, utility, and initialization methods. We do not recommend editing the generated methods. If you need to edit Javadoc comments in the generated files, see “Generating Javadoc” on page 9-16.



All changes to these files will be lost if you regenerate Java class wrappers for the same interfaces.

Generating Javadoc

Javadoc can be helpful for the Java developer who uses your JAR file. Javadoc is constructed from comments in the source files. To create Javadoc for your JAR file, you will need to do the following:

1. Add documentation comments for methods, return values, and parameters.

We include comments for the *xfNetLink* Java utility methods in every class, but you must provide a description for each of your user-defined methods if you want to produce useful documentation.

2. Select the “Generate Javadoc” option in Workbench or use the **genjava -j** option when you generate classes.

The generated classes include the Javadoc tags (e.g., `@param`) regardless of whether you have specified that you want to produce Javadoc. When you select “Generate Javadoc” or use the **genjava -j** option, a command to create Javadoc is added to the batch file. (The batch file is run either manually or from Workbench to create the JAR file.)



xfNetLink Java uses the standard Javadoc utility included in Sun Microsystems’ Java Development Kit to produce basic documentation that includes all the public methods in the JAR file. Should you wish to produce more elaborate Javadoc, do not use the “Generate Javadoc” or **genjava -j** option. Instead, run the Javadoc utility separately. See the Javadoc section of the Sun website for more information:

<http://java.sun.com/j2se/javadoc/index.html>

To add documentation comments to your source files, do one of the following:

- ▶ If you are using the MDU to populate the SMC, include the comment text in the description fields in the MDU as you define your methods. There are fields for specifying method description, return value description, and parameter description. (See [“Creating New Methods” on page 2-22](#) and [“Defining Parameters” on page 2-28](#) for instructions on entering data in these fields.)
- ▶ If you are attributing your code and using the XML file output by **dbl2xml** to populate the SMC, include the comment text for methods, return values, and parameters in your Synergy source files. See [“Documentation Comments” on page 2-19](#).
- ▶ Edit the generated source files (**.java** files) to add Javadoc comments or to add additional formatting tags to create the desired output. Keep in mind, though, that any changes you make to the **.java** files will be lost if you regenerate class wrappers. See [“To manually add Javadoc comments”](#) below for instructions.

Assuming your comments are in the SMC, when you generate class wrappers, the descriptions in the SMC are included in the generated files as Javadoc comments. If a method does not have a description in the SMC, a “To Do” comment is inserted in the generated file, as shown in [“To manually add Javadoc comments”](#) below. If a return value description is not provided in the SMC, the return data type is used as the Javadoc comment. If a parameter description is not provided in the SMC, the parameter name, Java data type, and direction passed are used as the Javadoc comment.

For structures passed as parameters, the field description in the repository is used as the Javadoc comment for the property methods in the generated class file. (There’s a “set” method and a “get” method associated with each property. See [“Structure Classes” on page 9-15](#) for more information.) The description text from the repository is preceded with either “Sets” or “Gets”. If a field does not have a description in the repository, “Sets the property value” and “Gets the property value” are used as the Javadoc comments.

Several Javadoc HTML files will be produced when you build the JAR file in Workbench or run the batch file. In the package directory, there will be an index file and three other “navigation” files. In a subdirectory within the package directory, there will be files for each of the classes. See [“Deploying the Client” on page 10-1](#) for information on making the Javadoc files available to your Java developers.

Creating Java Class Wrappers

Generating Javadoc

► To manually add Javadoc comments

1. Open the `.java` file(s) and find the “To Do” comments. There is a “To Do” comment before every user-defined method for which there was no description text provided in the SMC.
2. Replace the line that reads “***To Do*** Add method description” with a description of the method. Edit the comments for the `@param` and `@return` tags if desired. In structure classes, edit the descriptions for the set and get methods if desired.
3. Save the `.java` file(s).

Calling Synergy Routines from Java

This chapter explains how to deploy the JAR file and other tools necessary to create a distributed application with Java and Synergy, configure the *xy*NetLink Java properties file, and then use the JAR file. It also includes information on Java connection pooling and a method reference for the utility methods included in your class wrappers.

Deploying Your Distributed Application

This section tells you what needs to be installed and configured on the *xy*ServerPlus machine and the client machine. You will need to refer to this section when deploying your JAR file so that your Java developer can use it, as well as when creating the installation program for your distributed application.

Deploying the Server

For instructions on deploying the server portion of your distributed application, see [“Deploying the Server” on page 3-44](#).

Deploying the Client

1. Verify that JDK 1.2 or higher is installed. (Note: When you are deploying at a customer site, you probably will not need the complete JDK, but you will need the Java Runtime Environment.)
2. Install *xy*NetLink Java.
3. Set your classpath to point to the *xfnljav.jar* file, *xercesImpl.jar* file, *xml-apis.jar* file, and the *xfNLJava* directory. See [“Setting the Classpath” on page 9-3](#).
4. Copy your JAR file to the client machine. If you’re creating a web application, put it in the location required by your web server. This will vary depending on which web server software and servlet container you are using. For other Java applications, put it in a location where Java can find it, and set your classpath if necessary.

Calling Synergy Routines from Java

Configuring the xfNetLink Java Properties File

5. If you created Javadoc, copy the Javadoc HTML files to the client machine. You will need to distribute **index.html**, which will be located in the package directory, and the HTML files in the subdirectory created within that package directory. In this subdirectory, there will be an HTML file for each class in the JAR file. (Note: The other HTML files created in the package directory do not provide any additional information or functionality.)

The HTML files can go in any convenient location—just be sure to maintain the directory structure so that the links from the index to the individual files will work.

The Javadoc is intended to be a resource for the Java developer using your JAR file. When deploying at a customer site, you probably will not need to include the Javadoc.

6. (optional) Configure the *xfNetLink* Java properties file (see next section). If you're creating a web application, put the properties file in the location required by your web server. This will vary depending on which web server software and servlet container you are using. For other Java applications, put it in the same directory as the Java application.
7. If you are using Java connection pooling, configure the pooling properties file, and then place it in the location required by your application. See [“Setting Up a Pooling Properties File” on page 10-24](#) for more information.
8. If you are using Java connection pooling, run the client application to populate the pool. This step is important when you are deploying at a customer site, as it ensures that the pool is ready to use when the first user requests a connection.

Configuring the *xfNetLink* Java Properties File



See [Appendix A](#) for complete listings of all configuration settings for *xfServerPlus* and *xfNetLink*.

The *xfNetLink* Java properties file enables you to specify

- ▶ the host name and port number as defaults.
- ▶ a range of ports that the `SWPConnect` object uses to communicate with *xfServerPlus*.
- ▶ debug options.
- ▶ various time-out values.

Using a properties file is optional. If you choose not to use one, you can use the “set” methods associated with each procedural class in your JAR file to specify these values. (See [“Using a Properties File vs. Using the “set” Methods” on page 10-5.](#)) The set methods always take precedence over any entries in the properties file.



If you are using Java connection pooling, the xfNetLink Java properties file is required. See [“Specifying the xfNetLink Java properties file to use” on page 10-27](#) for information specific to using the properties file with pooling.

The table below shows the settings that can be included in the xfNetLink Java properties file. Each is discussed in more detail on the following pages. Because this file is used by Java, these settings are case sensitive.

Settings in the xfNetLink Java Properties File		
Setting	Description	See page
xf_RemoteHostName	Machine where xfServerPlus is running	10-6
xf_RemotePort	Port that xfServerPlus is running on	10-6
xf_LocalMinPort ^a	Lower limit of port number range for the SWPConnect object to use	10-7
xf_LocalMaxPort ^a	Upper limit of port number range for the SWPConnect object to use	10-7
xf_DebugOutput	Boolean value turns on client-side logging.	10-8
xf_LogFile	Filename that logging output should be written to (rather than the screen)	10-8
xf_SessionRequestTimeout	Number of seconds xfNetLink should wait for acknowledgment from the connection monitor in xfServerPlus	10-9

Settings in the <i>xfNetLink</i> Java Properties File (continued)		
Setting	Description	See page
xf_SessionConnectTimeout	Number of seconds <i>xfNetLink</i> should wait for acknowledgment from the logic server in <i>xfServerPlus</i> when running in normal mode	10-9
xf_DebugSessionConnectTimeout	Number of seconds <i>xfNetLink</i> should wait for acknowledgment from the logic server in <i>xfServerPlus</i> when running in debug mode	10-9
xf_SessionLingerTimeout	Number of seconds <i>xfNetLink</i> should wait for a return from a remote call	10-10

- a. The port number range feature will be deprecated in a future version; we do not recommend using this setting.

Creating and Naming a Properties File

There is a sample *xfNetLink* Java properties file named **xfNetLnk.ini** included in the Examples directory that is part of the *xfNetLink* Java installation. You can edit and use the sample file or create your own file in a text editor.

If you name your properties file “xfNetLnk.ini” and place it in the directory that your Java application starts in or in the default location required by your web server and servlet container, you can use the default constructor when you instantiate an object.

If you choose to name your properties file something other than “xfNetLnk.ini”, you must use the “ini file” constructor and pass the name of the properties file when you instantiate an object.

For details on using the default and ini file constructors, see “[Instantiate an instance of a procedural class](#)” on page 10-12.



To include comments in the properties file, precede the comment with a number sign (#).

Using a Properties File vs. Using the “set” Methods

As mentioned above, you can specify properties either in the *xfNetLink* Java properties file or by using the set methods. It is also possible to combine the two methods, specifying some properties in the file and setting others with the set methods. We do not, however, recommend this approach because it can result in confusion and be difficult to troubleshoot.

The table below lists the *xfNetLink* Java properties file settings and their corresponding “set” methods. As you can see, there are three time-out settings available in the properties file for which there are no corresponding set methods. If you wish to use these properties, you *must* use a properties file.

Properties File Settings and Corresponding “set” Methods	
Properties file setting	Set method
xf_RemoteHostName	setxfHost()
xf_RemotePort	setxfPort()
xf_LocalMinPort ^a	setxfMinport() ^a
xf_LocalMaxPort ^a	setxfMaxport() ^a
xf_DebugOutput	setxfLogging()
xf_LogFile	setxfLogfile()
xf_SessionRequestTimeout	N/A
xf_SessionConnectTimeout	N/A
xf_DebugSessionConnectTimeout	N/A
xf_SessionLingerTimeout	setxfCallTimeout()

a. The port number range feature will be deprecated in a future version; we do not recommend using this setting.

If your client application uses JavaServer Pages, you should keep in mind that you can have only one *xfNetLink* Java properties file. Most servlet containers require that the properties file be located in the container’s root directory. Because all objects use the properties file from the root directory, if you wanted, for example, to use different host and port settings for development vs. production, you would need to use the set methods.

Calling Synergy Routines from Java

Configuring the xfNetLink Java Properties File

If your client application is a standard Java application, the properties file must be located in the directory from which the application is started. This means that you could have several Java applications located in different directories, each with its own properties file.

The set methods always override settings in the properties file. When you use the default constructor, it preloads the following settings from the **xfNetLnk.ini** file: `xf_RemoteHostName`, `xf_RemotePort`, `xf_LocalMinPort`, `xf_LocalMaxPort`, `xf_DebugOutput`, and `xf_LogFile`. After an object is instantiated, if any of these items are set with the corresponding set methods, the set method value will override the preloaded value. The preloaded values will continue to be used if there is not an override set with a set method. If your **xfNetLnk.ini** file includes time-out settings, those settings will be ignored once you use one of the set methods. Note that any values set with the set methods stay in effect only for that object. When you instantiate another object, it may use different values.

Specifying the Host Name and Port Number

Specifying the host name and port in the *xfNetLink* Java properties file enables your Java object to read them as defaults. Using the properties file for the host name or port number is optional; you can also set them as properties of the class. Although *where* you set the host name and port is a matter of choice, setting them is not; they *must* be set either in the properties file or with the set methods.

► To specify the host name and port as defaults

In the *xfNetLink* Java properties file, specify the host name of the machine on which the *xfServerPlus* service is running and the port number on which it is listening. The default port is 2356. The port number must be an integer. Neither value can be null.

For example:

```
xf_RemoteHostName = elmo
xf_RemotePort = 2440
```

Specifying a Port Number Range



This feature will be deprecated in a future version; we do not recommend its use.

You can specify a range of port numbers for *xfNetLink* to use. *SWPConnect* uses ports from this range to communicate with *xfServerPlus*. If you do not specify a range, ports in the range 1024 through 65534 are selected at random.

Note the following:

- ▶ The port numbers must be integers.
- ▶ Both the minimum and maximum port numbers must be greater than 1024.
- ▶ `xf_LocalMinPort` must be less than or equal to `xf_LocalMaxPort`.

Be sure to specify a range that is large enough. If you run out of port numbers, *SWPConnect* throws *xfJCWException* with the message “No available ports in range.” (If the client is *not* using Java class wrappers, *ConnectException* will be thrown.)



If your client application is a web application, and there is a firewall between your web server and the system running *xfServerPlus*, you must set up packet filter rules that allow packets to be exchanged by these two systems on the range of ports specified with `xf_LocalMinPort` and `xf_LocalMaxPort`.

▶ To specify a port number range for *SWPConnect*

In the *xfNetLink* Java properties file, specify the beginning and ending values for the port number range. The range is inclusive. For example:

```
xf_LocalMinPort = 1056  
xf_LocalMaxPort = 2056
```

These values tell *xfNetLink* to use ports 1056 through 2056, inclusive.

Specifying Logging Options

This option causes the IP address, port number, and packets sent and received to be printed to the screen or written to a file. In addition, methods and parameters will be written to a separate file (for Java class wrappers only). For details on logging see [“Client-Side Logging” on page 11-7](#).

By default, information is output to the screen. You can specify that it be written to file instead. If the specified file cannot be opened for writing, an error is displayed and the information is output to the screen.



If your client application uses JavaServer Pages, output cannot be displayed on screen; it must be written to file.

► To specify client-side logging

1. In the *xfNetLink* Java properties file, set the debug output property to true.

```
xf_DebugOutput = true
```

2. To specify that the output be written to a file instead of displayed on the screen, specify a filename. For example,

```
xf_LogFile = c://work//Myfile.log
```

will direct the packet logging to the file **Myfile.log** and the method and parameter logging to a file named **MyfileJCW.log**. These files will be created if they do not exist; if the files already exists, additional material is appended to the end. To place the files in a specific location, specify the full path name, using double slashes as shown in the example above. If you do not specify a path, the files are created in the current working directory.

Specifying Time-out Values

You can specify three types of time-outs in the *xfNetLink* Java properties file (see [figure 5-1 on page 5-4](#)).

- Request for session
- Connect session (for normal and debug use)
- Call (session communication)

Request for session time-out

The request for session time-out ('A' in [figure 5-1](#)) measures the time that xfNetLink will wait to receive the initial acknowledgment from the connection monitor within xfServerPlus. The connection monitor is responsible for accepting session requests from xfNetLink and signaling the logic server to start a session.

The request for session time-out is measured from the time xfNetLink sends the request for a connection to the time xfNetLink receives the acknowledgment back from xfServerPlus. The default value is 120 seconds.

► To specify a request for session time-out

In the xfNetLink Java properties file, specify a value in seconds for `xf_SessionRequestTimeout`. For example, to specify a 5-minute time-out, use

```
xf_SessionRequestTimeout = 300
```

If the session request time-out value is invalid (less than zero, blank, or alpha), the default of 120 seconds is used.

Connect session time-out

Once the initial socket communication is established ('A' in [figure 5-1](#)), the connection monitor signals the logic server to start a session. The connect session time-out ('B' in [figure 5-1](#)) is measured from the time that xfNetLink receives the acknowledgment from the connection monitor to the time it receives an acknowledgment from the session started by the logic server in xfServerPlus.

This time-out is set separately for normal operation and debug operation. The default value for normal operation is 120 seconds; for debug it is 600 seconds. You will probably want to set the debug time-out to a greater value than the normal time-out since you need to move from one machine to another when starting a debug session (see the note on [page 11-12](#)).

► To specify a connect session time-out

In the xfNetLink Java properties file, specify a value in seconds for `xf_SessionConnectTimeout` and/or `xf_DebugSessionConnectTimeout`. For example, to specify a 3-minute time-out for normal operation and a 6-minute time-out for debug operation, use

```
xf_SessionConnectTimeout = 180  
xf_DebugSessionConnectTimeout = 360
```

Calling Synergy Routines from Java

Configuring the xfNetLink Java Properties File

If either of the connect session time-out values is invalid (less than zero, blank, or alpha), the corresponding default value is used.



The `xf_DebugSessionConnectTimeout` setting applies only when you use the `debugInit()` and `debugStart()` methods to run an `xfServerPlus` session in debug mode.

Call time-out

The call (session communication) time-out ('C' in [figure 5-1](#)) measures the length of time that `xfNetLink` waits for a return from a remote call. This time-out is measured for *each* send–receive request between `xfNetLink` and `xfServerPlus`. The default value is 1800 seconds (30 minutes).

To change this value after an object is instantiated, use the `setxfCallTimeout()` method. See “[Setting a Call Time-Out](#)” on page 10-19. You can also set a call time-out value for `xfServerPlus`. See [SET_XFPL_TIMEOUT](#) on page 1-31.

► To specify a call time-out

In the `xfNetLink` Java properties file, specify a value in seconds for `xf_SessionLingerTimeout`. For example, to specify a 20-minute time-out, use `xf_SessionLingerTimeout = 1200`

If the call time-out value is invalid (less than zero, blank, or alpha), the default value is used.

Using Your JAR File

Once you've created and deployed your JAR file, it is ready for use in a JSP or Java application. This section contains information that the person using your JAR file for development needs to know.

The machine used by the Java developer must be properly configured. See [“Deploying the Client” on page 10-1](#).



If you are using Java connection pooling, follow the steps in [“Using Your JAR File with Connection Pooling” on page 10-22](#) instead of the steps in this section.

1. Associate the component with your project

Before using the JAR file, you'll need to associate it with your project in your development environment. The method for doing this depends on the environment. If you are using Workbench, first make sure your project is the active project. Then, select Project > Project Properties. On the Files tab, click the Add Files button, navigate to the JAR file, and add it.

You can view the methods in the JAR file using the object/class browser supplied with your development environment. In Workbench, go to the Symbols tab in the project toolbar and expand the Packages/Namespaces node under the Workspace node.

You'll see a node for your JAR file. Expand it to see the classes in your JAR file. Expand the classes to view your own methods, along with the *xy*NetLink Java utility methods. The utility methods enable you to establish a connection with *xy*ServerPlus, disconnect from *xy*ServerPlus, and perform other utility functions, such as setting properties and running a debug session. See the method reference on [page 10-35](#) for a complete list of the utility methods.

2. Import the necessary packages

The `import` statement enables you to more easily access the classes in the imported package.

For Java applications, include an `import` statement for your package so that you can more easily access the classes in your JAR file. You'll also need to import the CORBA package so that you can use the “holder” classes (`DoubleHolder()`, `LongHolder()`, etc.). These classes are required to be able to return parameters from Synergy.

Calling Synergy Routines from Java

Using Your JAR File

For JSP applications, you do not need to import your package, but you should import the CORBA package to access the holder classes.

For example, in a Java application:

```
import com.ABCComputers.ConsultPro.*;
import org.omg.CORBA.*;
```

In a JSP application:

```
<%@ page import="org.omg.CORBA.*" %>
```

3. Instantiate an instance of a procedural class

There are two constructors included in procedural classes: the default constructor and the “ini file” constructor.

The default constructor takes no parameters. There are two situations in which you should use it.

- ▶ You are using an *xfNetLink* Java properties file named **xfNetLnk.ini**, which you have placed in the directory that your Java application starts in or in the default location required by your web server and servlet container.
- ▶ You are using the “set” methods to specify the host, port number, port range (min and max ports), and logging options. If one of these properties is not set with a set method, and there is a properties file named **xfNetLnk.ini** present that includes the setting, the setting in the **xfNetLnk.ini** file will be used.

For example, to use the default constructor in a Java application:

```
AppLogin appLog = new AppLogin();
```

Or, in a JSP page:

```
<jsp:useBean id="appLog" scope="session"
  class="ConsultIt.AppLogin">
```

The “ini file” constructor enables you to pass a Java `String` naming a specific properties file. Use it when your properties file is not named **xfNetLnk.ini**. If you use this constructor, do not use the set methods.

For example, to use the ini file constructor:

```
AppLogin appLog = new AppLogin("myPropFile.ini");
```

4. Set properties for the client

If you are not using a properties file, use the set methods in the class to set the properties for *xfServerPlus* host name and port, a range of ports for the client to use, and logging options. If you are using a properties file, you can override individual settings in it by using the set methods. See the method reference on [page 10-35](#) for a complete list of the set methods.

You must specify the host name and port either with the set methods or in a properties file. The other properties are optional.

For example:

```
appLog.setxfHost("elmo");  
appLog.setxfPort(2356);  
appLog.setxfLogging(true);  
appLog.setxfLogfile("c:\\temp\\consult.log");
```

5. Connect to *xfServerPlus*

There are several ways to establish the connection with *xfServerPlus*.

- Use `connect()`. This is the recommended method. It enables you to make several calls using the same connection, and then disconnect. There are several advantages to using `connect()`: it offers improved performance; you can maintain state between calls; and you can share this type of connection with other objects (see next bullet).

When you use `connect()`, you are instantiating an `SWPConnect` object, which then becomes a property of your object. The `SWPConnect` object is the actual connection to your *xfServerPlus* machine.

For example:

```
appLog.connect();
```

- Share a connection. Two or more objects can share a connection. This method improves performance because several objects are sharing the same *xfServerPlus* session rather than each object making its own connection. To share a connection, you must first establish it with `connect()`, and then use the `getConnect()` and `shareConnect()` methods. These objects then share a reference to the `SWPConnect` object.

In the example below, we instantiate two new procedural classes and use the `connect()` method to establish a connection for one of them. We then instantiate the `xyz` object to hold the connection, and use the `getConnect()`

Calling Synergy Routines from Java

Using Your JAR File

method to get it. Finally, we call the `shareConnect()` method of the `consult` object and pass the `xyz` object. You can pass the same `xyz` object multiple times to share the connection among several objects.

```
AppLogin appLog = new AppLogin();
Consultant consult = new Consultant();
appLog.connect();
java.lang.Object xyz = null;
xyz = appLog.getConnect();
consult.shareConnect(xyz);
```

- Create the connection automatically. This is referred to as an implied connection. When you make a call using one of the Synergy methods in your JAR file, the connection is created automatically and then disconnected when the call is complete. This is the easiest method to use because it makes access to *x/ServerPlus* completely transparent. However, this method does not allow you to maintain state between calls and requires more overhead because a connection is opened and closed for each call. If you use this method and experience performance problems, you may want to use an explicit `connect()` instead.

In the example below, the `consult` object is instantiated, and then the `postCharge()` method is called, without first calling the `connect()` method.

```
Consultant consult = new Consultant();
consult.postCharge(charge, return_msg);
```

6. Invoke methods in the component

Make calls to your Synergy methods and pass the parameters. If you generated Javadoc for your JAR file, it will include the information necessary to use the methods, such as the parameter data types.



Parameters that were not flagged as required in the SMC were converted to required parameters when you generated the Java class wrappers because you must always pass all parameters in Java.

For example, to invoke the `login()` method in the `AppLogin` class:

```
String id = new String("MFranklin");
String password = new String("123abc");
appLog.login(id, password);
```

7. Disconnect from *xfServerPlus*

If you connected to *xfServerPlus* using the `connect()` method, you must disconnect using the `disconnect()` method. If you have multiple objects sharing a connection, *xfServerPlus* will not completely close the connection and release the license until all objects are disconnected.

For example:

```
appLog.disconnect();
```



See [Appendix D](#) for a complete JSP code sample.

Using Structures

Repository structures passed as parameters to your Synergy methods are included in your JAR file as classes. There will be a separate class for each structure and for each group. The fields in the repository structure will become properties of the class.

For example, say you have the following repository structure, which is passed as the third parameter to the `login()` method.

```
user
  fname      ,a25
  lname      ,a25
  maxrate    ,d18.2
  group      address ,a
    street    ,a20
    city      ,a20
    state     ,a2
    zip       ,d9
  endgroup
```

In your JAR file, you'll see a class named "User" with the properties FName, Lname, and Maxrate. (Note that the first letter of each field name is capitalized.) There will be a set and get method for each property, named with the property name (e.g., `setFname()` and `getFname()`). Use the set and get methods in the class to assign values to the properties and retrieve values from them.



By default, the properties are named with the repository field names. See ["Passing Structures as Parameters" on page 1-8](#) for details on overriding the default name.

Calling Synergy Routines from Java

Using Your JAR File

You'll also see a class named "Address" in the JAR file. ("Address" is the name of the field in the repository that references this group.) This class represents the group within User. The group class will also have set and get methods for accessing its properties (e.g., `setStreet()` and `getStreet()`). When you instantiate a new User object, the User constructor automatically instantiates a new Address object named `Address_str`. Consequently, under normal circumstances, you will not need to access the Address class directly. (Fields defined as struct data type in the repository are treated the same as group fields.)

If the group is defined as an array in the repository, the Address object instantiated by the structure class will be named `Address_ary`. See the example under [step 4 on page 10-17](#).

For additional information about structure classes, see ["Understanding the Generated Classes" on page 9-14](#). For more information on passing structures as parameters, and for details on how overlays are handled, see ["Passing Structures as Parameters" on page 1-8](#).

1. To access the properties, instantiate the structure as a new object. For example:

```
User user1 = new User();
```

In our example, this will also instantiate a new object named `Address_str`, which represents the address group.

2. If you're sending data to Synergy, use the class's set methods to assign values to the properties. For example:

```
user1.setFname("Mickey");  
user1.setLname("Franklin");  
user1.setMaxrate(150.00);
```

Use the group's set methods to assign values to the properties of the group class:

```
user1.Address_str.setStreet("2330 Gold Meadow Way");  
user1.Address_str.setCity("Gold River");  
user1.Address_str.setState("CA");  
user1.Address_str.setZip(956704471)
```

If the group is an array, use the set methods in the `Address_ary` object:

```
user1.Address_ary[0].setStreet("2330 Gold Meadow Way");  
user1.Address_ary[1].setStreet("2445 Alpha Lane");
```

3. Pass the structure object as a parameter when you call the method:

```
appLog.login(id, password, user1);
```


4. If data is being returned from Synergy, instantiate new objects to hold the data, and then use the get methods to obtain the data:

```
String first = new String();
String last = new String();
double rate = new double();
first = user1.getFname();
last = user1.getLname();
rate = user1.getMaxrate();
```

The procedure for obtaining data from the properties in the group class is similar. For example:

```
String street = new String();
street = user1.Address_str.getStreet();
```

Passing Binary Data

You can pass binary data, such as JPEG files, by using an ArrayList.

In the MDU, define the parameter as a “Binary (handle)” data type. (If you attribute your code, see [example G on page 2-18](#) for instructions on defining a binary handle.) Your Synergy server routine must declare the argument that receives the data as a memory handle (i4). *x/ServerPlus* will place the data in a memory area and pass the memory handle allocated to that area to your Synergy server routine. After the data has been returned to *x/NetLink*, *x/ServerPlus* will free the memory area.

In your Java code, if the parameter is defined as “in” only in the SMC, you will need to create a byte array and fill it with data, instantiate an ArrayList, and add the byte array as the first element of the list. Then, you can make the method call. For example:

```
MyJCW JCWinstance = new MyJCW();
JCWinstance.setxfHost("HostMachine");
JCWinstance.setxfPort(2356);
JCWinstance.connect();

byte[] inba = new byte[67000]; //Create the byte[]
. //Fill byte[] with data
.
.
ArrayList al = new ArrayList(); //Create an ArrayList
al.add(inba); //Add byte[] as first element of ArrayList
JCWinstance.BinaryArrayMethod(al); //Call XFPL method
JCWinstance.disconnect();
```

Calling Synergy Routines from Java

Using Your JAR File

For “out” only parameters, in your Java code you will need to instantiate an `ArrayList`, call the method, and then extract the byte array from the first element of the `ArrayList` using the `ArrayList.get()` method. For example:

```
MyJCW JCWinstance = new MyJCW();
JCWinstance.setxfHost("HostMachine");
JCWinstance.setxfPort(2356);
JCWinstance.connect();
    ArrayList al = new ArrayList();    //Create ArrayList
    JCWinstance.BinaryArrayMethod(al); //Call XFPL method
    byte[] rtnba = (byte[])al.get(0);  //Extract byte[]
                                      // from first element
JCWinstance.disconnect();
```

For “in/out” parameters, in your Java code you will need to create a byte array and fill it with data, instantiate an `ArrayList` and add the byte array as the first element of the list, and then call the method. When the data is returned, you will extract the byte array from the first element of the `ArrayList` using the `ArrayList.get()` method. For example:

```
MyJCW JCWinstance = new MyJCW();
JCWinstance.setxfHost("HostMachine");
JCWinstance.setxfPort(2356);
JCWinstance.connect();

    byte[] inba = new byte[67000];    //Create the byte[]
    .                                //Fill byte[] with data
    .
    .
    ArrayList al = new ArrayList();  //Create ArrayList
    al.add(inba); //Add byte[] as first element of ArrayList
JCWinstance.BinaryArrayMethod(al); //Call XFPL method
    inba = (byte[])al.get(0); //Extract byte[] from
                              // first element
JCWinstance.disconnect();
```

Setting a Call Time-Out



You must explicitly create a connection with the `connect()` method to use `setxfCallTimeout()`.

Use the `setxfCallTimeout()` method to set the call time-out value in seconds. The call time-out measures the length of time that the Java client waits for a return from a remote call to *xfServerPlus*. (This is ‘C’ in [figure 5-1 on page 5-4](#).) This time-out is measured for *each* send–receive request between the client and *xfServerPlus*. The default value is 1800 seconds (30 minutes).

The call time-out can also be set in the *xfNetLink* Java properties file. Using the `setxfCallTimeout()` method enables you to change the properties file setting after the object has been instantiated. Once the call time-out value has been set with `setxfCallTimeout()`, it will continue to be used for all subsequent calls in the current session until it is reset with another invocation of this method.

If you are using shared connections, you can call this method on any object that is sharing the connection. The new time-out setting will affect all objects that share that connection.

For example, to set the call time-out to 10 minutes, use

```
appLog.setxfCallTimeout(600);
```

For information on specifying a call time-out value in the properties file, see “[Call time-out](#)” on [page 10-10](#). To set a call time-out value for *xfServerPlus*, see [SET_XFPL_TIMEOUT](#) on [page 1-31](#).



When you use Java connection pooling, there is a special call time-out that applies only to the initialization method. For information, see “[Specifying time-out values](#)” on [page 10-29](#).

Writing to the *xfServerPlus* Log



You must explicitly create a connection with the `connect()` method to use `setUserString()` or `getUserString()`.

Use the `setUserString()` method to pass a user string that is written in the *xfServerPlus* log. This string is stored, and the value set can be retrieved with `getUserString()`. You can, for example, use this method to write the current client in the log.

To use this method, server-side logging must be turned on (see [“Using Server-Side Logging” on page 3-29](#)) and there must be an entry for the XFPL_LOG subroutine in the SMC. By default, XFPL_LOG is included in the SMC; see [“Using the *xfServerPlus* Application Program Interface” on page 1-30](#).

For example, if your logon routine stored the user name in a string called `username`, you could write it to the log like this:

```
appLog.setUserString(username);
```

You can then retrieve the string you set. Note that `getUserString()` only gets the string from `setUserString()`; it does not retrieve it from the *xfServerPlus* log. For example:

```
String abc = new String();  
abc = appLog.getUserString();
```

Understanding Java Connection Pooling

Java connection pooling enables you to create a “pool” of connections to *xfServerPlus* that are active and ready to use when a client sends a request. You can specify the minimum and maximum pool size, time-out values, and whether connections in the pool should be reused. Depending on the requirements of your application, pooling can significantly improve performance by reducing the time necessary to establish connections and perform initialization processing. Pooling is of most benefit in JSP applications and in other Java applications, such as J2EE™ applications, in which multiple users access a single Java Virtual Machine running on a server.

How pooling works

When the pool starts up, it reads values from a pooling properties file to determine the configuration of the pool. The pool is populated with the minimum number of connections that you specify in the pooling properties file. When a client requests a connection, the request is satisfied from the connections available in the pool. Any Java class wrapper object that calls the `usePool()` method can use a connection from the pool. The connections in the pool do not time out. If no connection is available, a new one is created, up to the maximum size of the pool. Once the maximum is reached, requests can be queued for a specified length of time.

Reusing or discarding connections

When the client releases a connection, you can specify that it be either returned to the pool or discarded. By default, all connections are discarded after use. You can specify that connections be returned to the pool for reuse by setting an option in the pooling properties file.

In general, stateless connections may be returned to the pool, while connections with state (that is, those that persist data) should be discarded after use. The Synergy process that is attached to the connection may have state depending on whether the object that used the connection had state. Discarding a connection releases resources and ensures that the next client request receives a “clean” connection. If you decide to reuse connections, you can perform cleanup processing with the pooling support methods. If discarding a connection causes the pool to fall below the minimum level, a new connection is created.

Error logging

Using an option in the pooling properties file, you can specify that any errors that occur during pooling be written to a log file. You can specify that informational messages be logged as well. See [“Specifying logging options” on page 10-28](#) for more information.

Implementing Pooling

This section explains how to use your generated JAR file with connection pooling, how to set up and use a pooling properties file, and how to use the pooling support methods.

Using Your JAR File with Connection Pooling

This section contains information that the person using your JAR file for development needs to know. The machine used by the Java developer must be properly configured. (See [“Deploying the Client” on page 10-1](#) for details.) The code examples in this section use JSP, since that is the primary use for Java connection pooling.

1. Associate the component with your project

Before using your JAR file, you'll need to associate it with your project in your development environment. The method for doing this depends on the environment. See [step 1 on page 10-11](#) for details.

2. Import the necessary packages

The `import` statement enables you to more easily access the classes in the imported package.

You will need to import the CORBA package so that you can use the “holder” classes (`DoubleHolder()`, `LongHolder()`, etc.). These classes are required to be able to return parameters from Synergy.

We recommend that you import the `Synergex.util` package to make it easier to access methods of the `SWPManager()` class, which are required for pooling. (The example code in this section assumes this package is imported.)

For example:

```
<%@ page import="org.omg.CORBA.*" %>
<%@ page import="Synergex.util.*" %>
```

3. Create the pool

The `SWPManager.getInstance()` method is used to instantiate an instance of the pool at application level. The `SWPManager` class uses a *singleton* pattern. This means that there can be only a single instance of the class and that it is global by

nature. Consequently, the first time your application calls `getInstance()`, it will *instantiate* an instance of the pool. Subsequent calls to `getInstance()` will *get* an instance of the pool for use on a particular JSP page.

The `getInstance()` method uses the settings in the pooling properties file and the `xyNetLink` Java properties file to instantiate the pool(s) of connections. For JSP, you should put the initial call to `getInstance()` in your application's start-up page because the pool creation process may take a few moments. (The minimum number of connections must be created, initialization methods run, cleanup methods registered, etc.)

For example:

```
SWPManager poolMgr = SWPManager.getInstance();
```

4. Get an instance of the pool

Once the pool is created, you must get an instance of it in each JSP page that uses connection pooling. As explained above, after the initial call to `getInstance()`, subsequent calls will get an instance of the pool to use in a particular page.

For example:

```
SWPManager poolMgr = SWPManager.getInstance();
```

5. Instantiate an instance of a procedural class

Use the default constructor to instantiate a new instance of a procedural class in your generated JAR file. (Do not use the “ini file” constructor with pooling.)

For example:

```
<jsp:useBean id="appLog" scope="session"  
  class="ConsultIt.AppLogin">
```

6. Indicate that pooling will be used

Call the `usePool()` method on your object to indicate that this object will use a connection from the pool. Pass as parameters the ID of the pool that you want to get a connection from and the instance of the pool manager. The pool ID (“custPool” in our example) is defined in the pooling properties file. (See [“Setting Up a Pooling Properties File” on page 10-24](#).) The `usePool()` method is one of the public utility methods that is included when you build your JAR file.

For example:

```
appLog.usePool("custPool", poolMgr);
```

7. Get a connection, invoke methods in the component, disconnect

Call the `connect()` method to get a connection from the pool. You can then make calls to your Synergy methods, and call `disconnect()` when you are done. This part of your application is coded the same whether you are using pooling or not. You can also use an implied connection or share connections when you are using pooling. See [step 5 on page 10-13](#) for more information about the different ways to connect.

For example:

```
appLog.connect();
String id = new String("MFranklin");
String password = new String("123abc");
appLog.login(id, password);
.
.
.
appLog.disconnect();
```

Setting Up a Pooling Properties File

The pooling properties file defines settings, such as the size of the pool, used by Java connection pooling. This file is required. It must be named **xfPool.properties** and placed in either the Java application directory or in the directory required by your web server and servlet container.

There is a sample pooling properties file included in the Examples directory that is part of the *xyNetLink* Java installation. You can either copy the sample file to the correct location and modify it, or you can create your own text file, name it **xfPool.properties**, and put it in the correct location.

The pooling properties file supports multiple pools, each of which is identified by a pool ID. The file must identify at least one, named pool using the syntax

```
poolID.pool
```

where *poolID* is the name of the pool. For example:

```
orderPool.pool
```

The pool ID must precede each pool-specific setting in the file. (Logging level and log filename are not pool-specific; all pools use a single log file.) See the sample pooling properties file on [page 10-30](#).

Why use multiple pools?

By using multiple pools, you can specify that different pooling support methods be called for different objects. You may even have some objects that require no pooling support methods. For example, you might have a customer object and an order object, which require different initialization methods. You can create two pools, `custPool` and `orderPool`, and reference the appropriate initialization method ID for each pool. Then, when calling the `usePool()` method on the customer object, pass the `custPool` ID; and when calling the `usePool()` method on the order object, pass the `orderPool` ID.

Using multiple pools also enables you to have multiple *xfNetLink* Java properties files with different settings in each file. This will enable you to, for example, run *xfServerPlus* on more than one port or on different machines.



To include comments in the pooling properties file, precede the comment with a number sign (#).

The table below shows the settings that can be included in the pooling properties file. Each is discussed in more detail in the following pages. With the exception of the two logging settings, each setting in the pooling properties file must be preceded by the pool ID. There is a sample properties file on [page 10-30](#).

Settings in the Pooling Properties File		
Setting	Description	See page
<code>minPool</code>	The minimum number of connections to be maintained in the pool. Required.	10-26
<code>maxPool</code>	The maximum number of connections to be maintained in the pool. Required.	10-26
<code>propertiesFile</code>	The path and filename of the <i>xfNetLink</i> Java properties file to use. Required.	10-27
<code>poolReturn</code>	Boolean value that indicates whether the connection should be discarded or returned to the pool for reuse.	10-27
<code>poolLogFile</code>	Filename that logging information should be written to.	10-28

Calling Synergy Routines from Java

Implementing Pooling

Settings in the Pooling Properties File (continued)		
Setting	Description	See page
poolLogLevel	Level of logging desired. Possible values are none, error, and all. If poolLogFile is specified and poolLogLevel is not specified, error level logging will take place.	10-28
connectWaitTimeout	Number of seconds that the getConnection() method will wait for a connection from the pool.	10-29
initializationTimeout	Number of seconds that xfnNetLink should wait for a return from a remote call to xfServerPlus when the pool is being started. This value controls a special call time-out that applies only to the method specified with initializationMethodID.	10-29
initializationMethodID	Method ID of the Synergy method that will be called each time a new connection is added to the pool.	10-29
activationMethodID	Method ID of the Synergy method that will be called each time getConnection() is called.	10-29
deactivationMethodID	Method ID of the Synergy method that will be called each time a connection is freed.	10-29
poolableMethodID	Method ID of the Synergy method that will be called after the method specified by deactivationMethodID is called.	10-29
cleanupMethodID	Method ID of the Synergy method that will be called each time a connection is discarded (i.e., not returned to the pool for reuse).	10-29

Specifying the pool size

You must specify the minimum and maximum size of each pool defined in the pooling properties file. The `minPool` and `maxPool` settings are required.

Use `minPool` to indicate the number of connections you want created at pool start-up. The pool will never drop below this size. The minimum value for `minPool` is 1.

Use `maxPool` to indicate the maximum number of connections that you want in the pool at any one time. When deciding how large to make the pool, keep in mind how many *xfServerPlus* licenses you have available and the number of connections in all the pools. The maximum size of all connection pools should not exceed the number of available licenses.

Remember to precede `minPool` and `maxPool` with the pool ID. For example:

```
poolID.minPool=5  
poolID.maxPool=20
```

Specifying the *xfNetLink* Java properties file to use

When using Java connection pooling, you must use an *xfNetLink* Java properties file to specify information such as the host name and port. Using the “set” methods that were included when you built your JAR file will have no effect. Use the `propertiesFile` setting in the pooling properties file to specify the filename and location of the *xfNetLink* Java properties file you will be using. This setting is required.

When you are using pooling, you can name the *xfNetLink* Java properties file anything you like and place it anywhere on the *xfNetLink* Java machine because the setting in the pooling properties file enables *xfNetLink* to find it. In addition, you can have more than one *xfNetLink* Java properties file (one per pool, if desired), even for a JSP application.

For example:

```
poolID.propertiesFile=c:\\tomcat\\conf\\poolxfnj.ini
```

Specifying whether connections should be returned to the pool

When a connection is released, it can be either returned to the pool for reuse or discarded. See [“Reusing or discarding connections” on page 10-21](#) for more information on when connections can be reused. Set `poolReturn` to `true` to indicate that connections should be returned to the pool. Set `poolReturn` to `false` to indicate that they should be discarded. For example:

```
poolID.poolReturn=true
```

If `poolReturn` is not set, connections will be discarded.

Calling Synergy Routines from Java

Implementing Pooling

You can also control whether connections are returned to the pool at runtime by writing a “poolable” method and specifying it with the `poolableMethodID` property. See [“Specifying the pooling support methods to call” on page 10-29](#) and [“Pooling support methods” on page 10-33](#).

Specifying logging options

You can specify that actions related to the creation and maintenance of the pool be logged. All pools use a single log file.

► To turn logging on

1. Specify a log filename with the `poolLogFile` property. For example:

```
poolLogFile=c:\\work\\Myfile.log
```

Myfile.log is created if it does not exist; if the file already exists, additional material is appended to the end. To place the file in a specific location, specify the full path name, using double slashes as shown in the example above. If you do not specify a path, the file is created in the current working directory.

2. Specify a logging level with the `poolLogLevel` property. The available options are as follows:
 - **error.** Only errors and exceptions will be logged.
 - **all.** Informational messages, such as those generated when the pool is started, will be logged in addition to errors and exceptions
 - **none.** No logging will take place. Use this option to turn logging off without removing the logging settings from the pooling properties file.

For example:

```
poolLogLevel=all
```

If `poolLogFile` is specified and `poolLogLevel` is not specified, error level logging will take place.

Specifying time-out values

There are two time-out values that can be specified in the pooling properties file, `connectWaitTimeout` and `initializationTimeout`.

- ▶ The `connectWaitTimeout` value determines how long the `SWPManager.getConnection()` method will wait for a connection from the pool. If a connection is available, it will return immediately. If no connection is available and the pool is at its maximum size, `getConnection()` will wait the specified number of seconds for a connection to become available before returning null. If `connectWaitTimeout` is not specified, `getConnection()` will not wait, and will immediately return null.
- ▶ The `initializationTimeout` value determines the call time-out for the initialization method (i.e., the method specified with `initializationMethodID`). It replaces the `xf_SessionLingerTimeout` value only for the initialization method. If `initializationTimeout` is not specified, the `xf_SessionLingerTimeout` value in the *xfNetLink* Java properties file will be used. If `xf_SessionLingerTimeout` is not specified, the default value of 120 seconds will be used. (See [“Call time-out” on page 10-10](#) for more information on `xf_SessionLingerTimeout`.)

▶ To specify a connect wait time-out

Specify a value in seconds for `connectWaitTimeout`. For example, to specify a 1 minute time-out:

```
poolID.connectWaitTimeout=60
```

▶ To specify an initialization method time-out

Specify a value in seconds for `initializationTimeout`. For example, to specify a 3 minute time-out:

```
poolID.initializationTimeout=180
```

Specifying the pooling support methods to call

Java connection pooling supports the use of five pooling support methods. These are Synergy methods that you write, which are then called automatically at certain points during the lifetime of the connection pool. Using the pooling support methods is optional. (See [“Using the Pooling Support Methods” on page 10-32](#) for more information on writing the methods.)

Calling Synergy Routines from Java

Implementing Pooling

To use the pooling support methods, you must specify them by method ID—not method name—in the pooling properties file.

For example:

```
poolID.initializationMethodID=poolInit
poolID.activationMethodID=poolActivate
poolID.deactivationMethodID=poolDeactivate
poolID.cleanupMethodID=poolClean
poolID.poolableMethodID=poolReuse
```

Sample pooling properties file

The sample pooling properties file below includes logging settings that will be applied to all pools, along with settings for two pools. For “orderPool”, we specified all settings. For “custPool”, we specified only the required settings; default values will be used for all unspecified settings.

```
# Property settings for all pools
poolLogLevel=all
poolLogFile=c:\\tomcat\\logs\\Myfile.log

# Property settings for order pool
orderPool.pool
orderPool.minPool=2
orderPool.maxPool=6
orderPool.poolReturn=true
orderPool.connectWaitTimeout=45
orderPool.propertiesFile=c:\\tomcat\\conf\\poolxfnj.ini
orderPool.initializationTimeout=60
orderPool.initializationMethodID=poolInit
orderPool.activationMethodID=poolActivate
orderPool.deactivationMethodID=poolDeactivate
orderPool.cleanupMethodID=poolClean
orderPool.poolableMethodID=poolReuse

# Property settings for customer pool
custPool.pool
custPool.minPool=4
custPool.maxPool=10
custPool.propertiesFile=c:\\tomcat\\conf\\poolxfnj.ini
```

Pool Maintenance

The `SWPManager` class includes several methods that you can use to change the pool configuration: `resetPoolProperties()`, `returnToMinimum()`, and `shutdown()`. We recommend that you write a separate utility program or JSP page that calls these methods and can be accessed by the system administrator.

Updating the properties files

You may need to change settings in the pooling properties file or the *x/NetLink* Java properties file after the application has been started. Once you have updated the properties file(s), call the `resetPoolProperties()` method from your administrative page, passing the ID of the pool to update. This method closes down all of the available connections in the specified pool and then restarts the pool with the minimum number of connections. The new settings in the pooling properties file are read when the pool is restarted, as are the settings in the *x/NetLink* Java properties file used by that pool. For more information about `resetPoolProperties()`, see the `SWPManager` class on [page 12-55](#).

Alternatively, you can stop and restart the application to update the properties files. To do this, you have to shut down your web server or servlet container. The new pool will be created using the new values.

Returning the pool to the minimum size

You can call the `returnToMinimum()` method to return a specific pool to the size specified with `minPool`, thereby freeing up *x/ServerPlus* licenses so that they can be used by other pools. For example, when `poolReturn` is set to true, the pool may grow to the maximum size and then, because all connections are being returned to the pool, remain at the maximum. (When `poolReturn` is set to false, the pool will always be maintained at the minimum size, so calling `returnToMinimum()` will have no effect.) Rather than including this method in a separate utility program or JSP page that can be accessed by the system administrator, you may want to call it periodically in your application.

For more information about `returnToMinimum()`, see the `SWPManager` class on [page 12-55](#).

Shutting down the pool

The `shutdown()` method enables you to shut down all connection pools. Call this method when closing down your application.

When you call the `shutdown()` method, all available connections are closed and the pools are destroyed. Connections that are in use when `shutdown()` is called will be terminated when they are released. The deactivation and cleanup methods will not be called on those methods. (Cleanup is called on the connections that are in the pool when it is shut down.) The next call to `getInstance()` will restart the pools.

Using the Pooling Support Methods

Java connection pooling supports five optional user-defined methods that are called automatically at specified times during the life of the pool. These methods enable you to specify that certain actions be performed at specified times during the connection's creation and use.

► To use the pooling support methods

1. Write Synergy routines that perform the desired tasks. You can have several routines of the same type, if necessary (e.g., two initialization routines for two different pools), and the routines can be named anything you like. See [Pooling support methods on page 10-33](#) for the syntax your routine should use, a description of the purpose of each method, and when each method is called.
2. Use the Method Definition Utility to add the routines to the SMC. You do not need to specify an interface name, as these routines will not be included in the generated JAR file. (Because you do not call these methods directly, there is no reason to include them in the JAR file.)
3. Include the Synergy routines when you build your ELB. You can put them in the same ELB as the other Synergy routines you prepared for remote calling, or you can put them in a separate ELB.
4. Specify the method ID—not the method name—of each routine in the pooling properties file, using the property that corresponds to the type of routine. See [“Specifying the pooling support methods to call” on page 10-29](#).

Pooling support methods

The methods are described below in the order in which they are called during a connection's lifetime.

Initialization method

status = initialization_method()

status—a ^VAL value that indicates whether the initialization method was successful. Returns 0 for success or 1 for failure. If the return value is 1, the pool will not be created, `xfPoolException` will be thrown, and an error will be recorded in the pooling log (assuming logging is turned on).

This method is called each time a new connection is added to the pool. You can use this method to prepare the environment by opening files, initializing global data, and so forth. Because the initialization method is called when the connection is created, it gets called only once per connection, even if the connection is returned to the pool for reuse. Compare with the activation method.

Activation method

activation_method()

The activation method is called when the connection is requested by a client. This method can be used for code that should be executed when the connection is actually used.

Both the activation and the initialization methods can be used for similar purposes—preparing the environment before using the connection. The primary difference between them is that the activation method is called *every* time the connection is allocated to a client, whereas the initialization method is called only once when the connection is added to the pool.

Deactivation method

deactivation_method()

The deactivation method is called when a connection is released. It can be used to reset the environment to a known state before a connection is returned to the pool. Because connections can be reused, this method may be called numerous times. Compare with the cleanup method.

Poolable method

status = poolable_method()

status—a ^VAL value that indicates whether the connection should be discarded or reused. Returns 0 if the connection should be discarded; returns 1 if the connection should be returned to the pool for reuse.

The poolable method is called after the deactivation method and can be used to determine at runtime if a connection should be returned to the pool or discarded. For example, if the deactivation method encounters an error, the poolable method could check how much effort is required to clean up the connection before returning it to the pool. If the effort is excessive, and it would be more efficient to discard the connection and create a new one, the poolable method would return 0. See [“Reusing or discarding connections” on page 10-21](#) for more information on when connections can be reused.

The poolable method overrides the `poolReturn` setting in the pooling properties file.

Cleanup method

cleanup_method()

The cleanup method is called each time a connection is discarded from the pool. This includes when `poolReturn` is set to false, when the poolable method returns 0, or when any of the following methods in the `SWPManager` class are called: `shutdown()`, `resetPoolProperties()`, `returnToMinimum()`. The cleanup method can be used to do final cleanup on the connection. If the connection is going to be reused, use the deactivation method instead to perform cleanup-type activities.

The cleanup method is also called when socket communication with the client is unexpectedly lost. When the pool is created, the cleanup method is automatically registered with the `XFPL_REGCLEANUP` routine on the server. (This routine must be in your SMC; see [page 1-34](#).) Then, if there is a fatal error that causes `xfServerPlus` to lose socket communication with the client, `xfServerPlus` calls the cleanup routine before shutting down.

Method Reference

This is a listing of the public utility methods included in all procedural classes in your Synergy JAR file.

connect()

public void connect() throws xfJCWException

Sends a request to *xfServerPlus* for a dedicated connection and connects on the host and port defined in the *xfNetLink* Java properties file or specified with *setxfHost()* and *setxfPort()*. See [“Connect to xfServerPlus” on page 10-13](#).

debugInit()

public void debugInit(String *clientIP*, StringBuffer *listeningIP*, StringBuffer *listeningPort*) throws xfJCWException

Initializes a connection to *xfServerPlus* so you can manually start an *xfServerPlus* session in debug mode. See [“Running an xfServerPlus Session in Debug Mode” on page 11-11](#).

clientIP—the IP or name of the *xfNetLink* Java client machine

listeningIP—returns the IP address, in hex, where the client is listening

listeningPort—returns the port number where the client is listening

debugStart()

public void debugStart(String *clientIP*) throws xfJCWException

Completes the process of connecting in debug mode that was started with *debugInit()*. See [“Running an xfServerPlus Session in Debug Mode” on page 11-11](#).

clientIP—the IP or name of the *xfNetLink* Java client machine

disconnect()

public void disconnect() throws xfJCWException

Sends a message to *xfServerPlus* to close the connection. See [“Disconnect from xfServerPlus” on page 10-15](#).

getConnect()

public Object getConnect() throws xfJCWException

Gets the already-established connection of the specified object. See [“Connect to xfServerPlus” on page 10-13](#).

getUserString()

```
public String getUserString()
```

Returns the string currently stored by `setUserString()`. See [“Writing to the xfServerPlus Log” on page 10-20](#).

getxfHost()

```
public String getxfHost()
```

Returns the host name that was set with `setxfHost()`.

getxfLogfile()

```
public String getxfLogfile()
```

Returns the name of the client-side log file, which was set with `setxfLogfile()`.

getxfMaxport()



This method will be deprecated in a future version; we do not recommend its use.

```
public int getxfMaxport()
```

Returns the port number that was set with `setxfMaxport()`.

getxfMinport()



This method will be deprecated in a future version; we do not recommend its use.

```
public int getxfMinport()
```

Returns the port number that was set with `setxfMinport()`.

getxfPort()

```
public int getxfPort()
```

Returns the port number that was set with `setxfPort()`.

setUserString()

```
public void setUserString(String userString) throws  
xfJCWException
```

Passes a user string to the *xfServerPlus* log. See [“Writing to the xfServerPlus Log” on page 10-20](#).

userString—the text that you want to write to the *xfServerPlus* log

setxfCallTimeout()

public void setxfCallTimeout(int *seconds*) throws xfJCWException

Sets the call time-out value, which measures the length of time that the client waits for a return from a remote call. See [“Setting a Call Time-Out” on page 10-19](#).

seconds—the number of seconds you want the *xfNetLink* Java client to wait for a return from a call to *xfServerPlus*. The default is 1800 seconds (30 minutes).

setxfHost()

public void setxfHost(String *hostName*)

Specifies the host name of the server machine.

hostName—the IP address or host name of the machine where *xfServerPlus* is running

setxfLogfile()

public int setxfLogfile(String *logFilename*)

Specifies the name of the log file for client-side logging. If you do not specify a log file, data is displayed on the client screen. See [“Client-Side Logging” on page 11-7](#).

logFilename—the filename to which you want the client-side packets written. Include the full path if desired; otherwise, the file will be placed in the current directory. If the file does not exist, it is created; if it already exists, additional material is appended to the end. The additional Java class wrappers logging is written to a file named with this name, with JCW appended to the end.

setxfLogging()

public int setxfLogging(boolean *logging*)

Turns on client-side logging, which enables you to view packets and input and output parameters on the client side (also referred to as debug trace output). See [“Client-Side Logging” on page 11-7](#).

logging—Boolean value to indicate that logging is on. Set this to true to activate logging.

setxfMaxport()



This method will be deprecated in a future version; we do not recommend its use.

`public void setxfMaxport(int port)` throws `xfJCWException`

Sets the ending value for the port number range that the `SWPConnect` object uses.

port—the ending value for the port number range. This value must be greater than 1024 and greater than or equal to the value set by `setxfMinport()`. If no port range is specified, ports in the range 1024 through 65534 are selected at random.

setxfMinport()



This method will be deprecated in a future version; we do not recommend its use.

`public void setxfMinport(int port)` throws `xfJCWException`

Sets the beginning value for the port number range that the `SWPConnect` object uses.

port—the beginning value for the port number range. This value must be greater than 1024 and less than or equal to the value set by `setxfMaxport()`. If no port range is specified, ports in the range 1024 through 65534 are selected at random.

setxfPort()

`public void setxfPort(int port)`

Sets the port number of the machine on which `xfServerPlus` is running.

port—the port number that `xfServerPlus` is listening on for remote session requests. Must in the range 1024 through 65534. The default is 2356.

shareConnect()

`public void shareConnect(Object connection)` throws `xfJCWException`

Shares the specified connection. See [“Connect to xfServerPlus” on page 10-13](#).

connection—the object that represents the connection

usePool()

```
public void usePool(String poolID, SWPManager poolManager)
```

Indicates that the object will be using Java connection pooling.

poolID—the ID of the pool to use

pool—the instance of the pool manager to use. This is instantiated with the `SWPManager.getInstance()` method. For more information on `getInstance()`, see the `SWPManager` class on [page 12-55](#).

Error Handling and Troubleshooting in *xfNetLink* Java

This chapter includes a table that lists the exceptions that can be thrown when using *xfNetLink* Java Edition and what you can do to resolve them. It also includes information on troubleshooting, running the *xfNetLink* Java test program, and logging. For a full discussion of exception handling in Java, consult your Java documentation.

Handling Exceptions

Because applications built using *xfNetLink* Java and *xfServerPlus* include multiple programs distributed across multiple machines, error handling design must take into account the possibility of errors occurring in any one of the programs.

Error conditions can occur on either the *xfNetLink* or *xfServerPlus* side of a remote execution session. In a distributed environment, error handling requires multiple layers of processing: the error may occur on a machine that the user cannot see, and the user may not know how to contact the owner of the machine that failed.

To help you deal with this situation, errors detected by *xfServerPlus* are always logged to the application event log (Windows), syslog (UNIX), or operator console (OpenVMS). Errors may also be logged to the *xfServerPlus* log, if you have server-side logging turned on. (See [“Using Server-Side Logging” on page 3-29](#).) This enables the system administrator to track problems that have occurred even if they are not reported by a user. In addition, those same errors are reported to the *SynergyWebProxy* or *SWPConnect* object so that your application can handle the error and, when necessary, inform the user about the problem. Note that by default, no logging occurs on the client.

Exceptions in xfNetLink Java

The table below lists exceptions that may occur while running xfNetLink Java, what causes them, and what you can do to resolve the problem. Exceptions that are part of the Java classes included with xfNetLink Java are also documented in [chapter 12](#).

If your code uses Java class wrappers, all errors related to xfNetLink Java and xfServerPlus are thrown as xfJCWException, which is a wrapper class for the other xfNetLink Java exceptions. For optimum error handling, your code should catch and handle xfJCWException. If you are using Java class wrappers with Java connection pooling, you should also catch xfPoolException.

Exceptions in xfNetLink Java			
Exception	Cause	What to do	Comments
BadSynPacketException	Method ID is null.	Correct your code to include the method ID.	getMessage() will tell you that the method ID is missing.
	Noise on the line results in a corrupt packet being sent or received.	Retry. If the problem persists, contact your system administrator.	You can examine the packet contents to see if it was sent and received correctly.
	Packet is corrupted.	Call Synergy/DE Developer Support if the problem persists.	This error will probably not be logged because the information was probably sent correctly as far as xfServerPlus could tell.
ConnectException	All ports in the range specified in the xfNetLink Java properties file have been used.	Increase the size of the port number range.	See “Specifying a Port Number Range” on page 10-7.
	xfServerPlus is not listening on the specified port.	Start xfServerPlus on the correct port or change the listening port specified in the xfNetLink Java properties file.	See “Running xfServerPlus” on page 3-3 or “Specifying the Host Name and Port Number” on page 10-6.

Error Handling and Troubleshooting in xfNetLink Java

Exceptions in xfNetLink Java

Exceptions in <i>xfNetLink</i> Java (continued)			
Exception	Cause	What to do	Comments
InterruptedException	<i>xfNetLink</i> times out on a send or receive.	Increase the session or call time-out value. If the problem persists, contact your system administrator.	See “Request for session time-out” on page 10-9 or “Call time-out” on page 10-10.
IOException	Network connection or socket goes down.	Retry. If the problem persists, contact your system administrator.	N/A
SynProxyBadProps Exception	Bad or missing port or host name in the <i>xfNetLink</i> Java properties file.	Correct the port and host name.	See “Specifying the Host Name and Port Number” on page 10-6.
SynProxyBadType Exception	Format of returned data is incompatible with format of sent data.	Correct your Java code to use a more appropriate SynergyType object as a parameter in the remote call.	Use getMessage() to learn details of the mismatched type.
	SynergyType object created with incorrect data (e.g., synDec("abc")).	Correct your Java code.	N/A
SynProxyIncompatible ServerException	The version of <i>xfServerPlus</i> is not compatible with the version of <i>xfNetLink</i> for the current operation.	Upgrade <i>xfServerPlus</i> to the current version.	The message will return the protocol version required. See the table on page 12-19 for how this maps to the software version.
SynProxyInvalidStruct Exception	Attempted to create a structure that is internally inconsistent. For example, the structure size does not match the sum of the defined field sizes.	Call Synergy/DE Developer Support.	N/A
SynProxyMixedTypeArray Exception	There's more than one type of SynergyType object in an array.	Use getMessage() to get a message identifying the element with the differing type; correct your code.	A synArray object can contain only one type of SynergyType object.

Error Handling and Troubleshooting in xfNetLink Java

Exceptions in xfNetLink Java

Exceptions in <i>xfNetLink</i> Java (continued)			
Exception	Cause	What to do	Comments
SynProxyNetException	<i>xfNetLink</i> timed out because of very slow start-up.	Increase the connect session time-out value. If the problem persists, contact your system administrator.	See “Connect session time-out” on page 10-9.
	(WIN, UNIX) The version of <i>xfServer</i> that is running does not support <i>xfServerPlus</i> .	Upgrade <i>xfServerPlus</i> to the current version.	N/A
	<i>xfServerPlus</i> was started without the remote execution option.	Restart <i>xfServerPlus</i> using the -w option.	The <i>xfServerPlus</i> session is not started; consequently, no information is logged.
xfJCWException	Numerous.	Read the error message to determine the specific problem.	<i>xfJCWException</i> serves as a wrapper for the other exceptions in xfnljav.jar (except <i>xfPoolException</i>). It is thrown only when your code uses Java class wrappers.
xfLaunchFatalException	Fatal error during execution of a Synergy routine caused by a non-trappable problem.	For additional information, see the <i>xfServerPlus</i> log or the event log (Windows), syslog (UNIX), or operator console (OpenVMS). Use the <i>xfLaunchFatalException</i> methods to obtain the Synergy Language and system error numbers and the name of the routine that failed. Make corrections to your code. Be sure to check the number and type of parameters.	The <i>xfServerPlus</i> session terminates and the socket collapses.

Error Handling and Troubleshooting in xfNetLink Java

Exceptions in xfNetLink Java

Exceptions in <i>xfNetLink</i> Java (continued)			
Exception	Cause	What to do	Comments
xfPoolException	Missing or incorrect data in the pooling properties file or the <i>xfNetLink</i> Java properties file.	Read the error message to determine which setting has the problem, and then correct the properties file.	This exception is thrown only when your code uses Java connection pooling.
	Problems creating the pool, including <i>xfServerPlus</i> licensing issues.	Read the error message to determine the specific problem.	The pooling log file can also help determine what the problem is. Some error conditions may not throw an exception because they occur after the connection has been terminated. They will, however, still be recorded in the log.
	Errors that occur when calling the pooling support methods, including errors within the Synergy method being called.	Read the error message to determine which method has a problem.	
xfRemoteExecException	<i>xfServerPlus</i> cannot translate a particular SynergyWebProxy xcall() into a Synergy routine call.	Use the <i>xfRemoteExecException</i> methods to obtain the error number, message, and supporting text. Use this information to determine if the error is in the Java code, Synergy code, or SMC. Make corrections to your code or the SMC.	The <i>xfServerPlus</i> session does not terminate. This exception may be caused by call creation errors such as failure to send a required parameter, sending the wrong number of parameters, or sending the wrong data type in a parameter.
	<i>xfServerPlus</i> has a problem (e.g., gets an ISAM error reading the SMC) or a routine that's being executed causes an error.	Use the <i>xfRemoteExecException</i> methods to obtain the error number, message, and supporting text. Use this information to determine if the error is in the Synergy code or was generated by a problem in <i>xfServerPlus</i> . Make corrections to your code.	N/A

Error Handling and Troubleshooting in xfNetLink Java

Exceptions in xfNetLink Java

Exceptions in <i>xfNetLink</i> Java (continued)			
Exception	Cause	What to do	Comments
xfServerNackException	<i>xfServerPlus</i> is not licensed.	Purchase a license.	The <i>xfServerPlus</i> session is not started; consequently, no information is logged. Use the methods associated with <i>xfServerNackException</i> to find out what errors were returned.
	(VMS) <i>xfServerPlus</i> is licensed but has exceeded the number of available licenses.	Upgrade the number of licenses.	
	<i>xfServerPlus</i> cannot access the license database.	Contact your system administrator.	
	(WIN, UNIX) No Synergy Runtime license available.	Verify there is a Runtime license on the server machine.	
	(WIN, UNIX) Invalid user name or password used to start <i>xfServerPlus</i> .	Verify the user name and password.	
	<i>xfNetLink</i> client version is incompatible with the <i>xfServerPlus</i> version.	Upgrade your client or your server to the higher version.	Using a version of the server that predates your client version is not a supported configuration.
	(WIN) The user account used to start <i>xfServerPlus</i> does not have the "log on as a batch job" user right set.	This error can occur only if you are using <i>xfServerPlus</i> version 7.1.5 or earlier. Upgrade <i>xfServerPlus</i> to the current version.	
xfServerPlusUnavailable Exception	(WIN, UNIX) <i>xfServerPlus</i> is licensed but has exceeded the number of available licenses.	Retry. If the problem persists, purchase additional licenses.	<i>xfServerPlus</i> starts, logs the error, sends a denial of service message, and terminates.
	(WIN, UNIX) The 14-day demo period or an extended demo period has expired.	Call your Synergy/DE customer service representative.	
	A serious error, such as inadequate memory, occurred while <i>xfServerPlus</i> was starting.	Retry. If the problem persists, contact your system administrator or Synergy/DE Developer Support.	

Troubleshooting Techniques

Error messages don't always provide enough diagnostic information to solve a problem. In such cases, you can take advantage of the additional debugging options provided with *xfNetLink* and *xfServerPlus*: client-side logging, the *xfNetLink* Java test program, and the ability to run an *xfServerPlus* session in debug mode.

xfNetLink Javaclient-side logging is not as extensive as the server-side logging available with *xfServerPlus*. See [“Using Server-Side Logging” on page 3-29](#) for more information. You may also want to run the test program included with *xfServerPlus*; see [“Testing *xfServerPlus*” on page 3-15](#).

Client-Side Logging

Client-side logging for *xfNetLink* Java produces two files. One contains the complete packets sent back and forth between the Java application and *xfServerPlus*. The other lists the methods called and their input and output parameters. (Note that this second file is produced only if you are using Java class wrappers.)

You can enable client-side logging either by setting values in the *xfNetLink* Java properties file (e.g., **xfNetLnk.ini**) or by calling a method in one of the classes in your JAR file.

By default, the file containing the packets is output to the client's screen. You can direct the output to a file instead by specifying a filename (see below for details). The file containing the packets is named with the specified filename, and the file containing the methods and parameters is named with the specified filename with JCW appended to the end. (Note that you must specify a filename to see the method and parameter information; only the packet information is output to the screen.)

► To enable logging by setting a value in the properties file

Set `xf_DebugOutput` to true in the *xfNetLink* Java properties file. To write the output to file, specify the filename with the `xf_LogFile` option. All processes will write to the same file. See [“Specifying Logging Options” on page 10-8](#) for more information on using the properties file settings.

Error Handling and Troubleshooting in xfNetLink Java

Troubleshooting Techniques

► To enable logging by calling a method

Call the `setxfLogging()` method and pass “true” as the parameter. To write the output to file, use the `setxfLogfile()` method and pass the filename. (`setxfLogging()` and `setxfLogfile()` are two of the utility methods included in each class in your JAR file.) The log files will contain data only from the class from which they are called. See the method reference on [page 10-35](#) for more information about `setxfLogging()` and `setxfLogfile()`.



If your client application uses JavaServer Pages, the output cannot be displayed on screen; you must write the output to file.

The packet output shows the IP address that SWPConnect is listening on, the current time-out settings, and the packets sent and received. As shown in the sample below, connection information prints once for each session; sent and received data prints for each call to *xfServerPlus*.

```
-----
SynergyWebProxy session beginning: Aug 20, 2009 10:15:18 AM PST
Local host: tiger/111.22.33.44
Request timeout = 120 seconds
Connect timeout = 600 seconds
Communicate timeout = 1800 seconds
-----

Sending string:
Jcompid0009;2;AL5#abcde;AL5#54321;
Received string:
Rcompid0009;03;00AL9#Return    ;01AL5#back1;02AL5#back2;

Sending string:
Jcompid0010;2;AL5#abcde;AL5#54321;
Received string:
Rcompid0010;02;00DE9#123456789;02AL5#back ;
```

The method and parameter output shows the IP address, followed by each method called and its input and output parameters. The parameter information includes the Java data type, parameter name, and the data in the parameter.

```
SynergyWebProxy session beginning: Aug 20, 2009 11:35:02 AM PST
Local host: tiger/111.22.33.44
** In method: function_one
  * Input parameters
    String p1 = This is an Alpha field 50 characters
              long a test
    long p2 = 12345
    double p3 = 1234567.911
    int p4 = 12345
  * Output parameters
** In method: function_two
  * Input parameters
    DoubleHolder p1 = 1.23456789012E9
    DoubleHolder p2 = 12345.6789
    double p3 = 12345.6789
    double p4 = 1.234567891E7
    long p5 = 1234567890
    DoubleHolder p6 = 0.123456
    double p7 = 1.2345
    LongHolder p8 = 12345678
    DoubleHolder p9 = 123456.7
    long p10 = 123456789
  * Output parameters
    DoubleHolder p1 = -1.23456789112E9
    DoubleHolder p2 = 333.334
    DoubleHolder p6 = 0.9988332
    LongHolder p8 = -88991010
    DoubleHolder p9 = 654321.0
```

Testing xfNetLink Java

The **xfNLJTest** program, distributed with **xfNetLink Java**, can help you determine if your system is set up and working properly.

xfNLJTest runs several tests, which call functions and send different types of data back and forth between the Java client and the Synergy server. This program makes calls to a test ELB or shared image named **xfpl_tst**, which is distributed with **xfServerPlus**. There are entries in the SMC for use by the test program. (These are the methods in the **xfTest** interface in the distributed SMC.) If the ELB or any of the methods are missing, the tests will fail.



If the methods in the `xfTest` interface are not present in your SMC, you can import them from the **defaultsmc.xml** file. See [“Importing and Exporting Methods” on page 2-39](#).

► To run the `xfNLJTest` program

1. Make sure `xfServerPlus` has been started on the server machine. (See [“Running `xfServerPlus`” on page 3-3](#).)
2. On the client machine, go to the directory in which you installed `xfNetLink Java` and type the following at the command line:

```
java xfNLJTest xfServerMachineName xfServerPort
```

For example:

```
java xfNLJTest elmo 2356
```

As the tests run, information is printed to the screen and saved to the **xfNLJtst.txt** file. This file is created in the directory from which you ran the test. You will see a line describing each test and a message stating whether the test completed successfully. The output also includes the versions of `xfNetLink` and `xfServerPlus` being used, which may be helpful in troubleshooting.

You may receive errors in the form of Java exceptions. The exception message text should help you determine what needs to be done to correct the problem. Check the table of exceptions starting on [page 11-2](#) and your Java documentation for assistance. Exceptions are also written to **xfNLJtst.txt**.

If any tests were unsuccessful, check **xfNLJtst.txt** for more information. You may want to also run the `xfServerPlus` test program, **xfsplstst**. It can help you determine whether the problem is on the `xfNetLink` side or the `xfServerPlus` side. See [“Testing `xfServerPlus`” on page 3-15](#) for more information.

If you cannot solve the problem, call the Synergy/DE Developer Support department. Be sure to save the **xfNLJtst.txt** file; your Support engineer needs the information in this file to help you.

Running an xfServerPlus Session in Debug Mode

During normal operation, *xfServerPlus* runs as a background process without support for console operations, complex user interfaces, or debugging. This improves efficiency and minimizes memory requirements. However, there may be times when you need to run the debugger on Synergy code in the ELBs or shared images that are being called from *xfServerPlus*. By manually connecting an *xfServerPlus* session to your running Java application, you can run your Synergy routines in debug mode so that you can uncover underlying problems that are showing up as errors in your distributed application.



We recommend that you use the Telnet method for debugging if the operating system of your *xfServerPlus* machine is Windows or UNIX. See [“Debugging Remote Synergy Routines via Telnet” on page 3-42](#) for instructions.

Running in debug mode on Windows and UNIX

Use this procedure if the operating system of your *xfServerPlus* machine is Windows or UNIX.

If your SMC files or **xfpl.ini** file are not in the default location (DBLDIR), you will need to either move them to DBLDIR or set XFPL_SMCPATH and XFPL_INIPATH *in the environment* to point to the location of the files before starting **xfpl.dbr** (step 4). (Note: When XFPL_SMCPATH and XFPL_INIPATH are set in the registry or **synrc**, they are read by **rsynd**. Since **rsynd** is bypassed when you run in debug mode, the registry/**synrc** settings do not get read.)



If your client application uses JavaServer Pages, you will need to use two JSP pages to run in debug mode. Use the first page to make the call to `debugInit()` and return the HTML that displays the IP and port. Then, you'll need a second page with the `debugStart()` call. After submitting the second page, the client will go into wait mode while you start *xfServerPlus*.

Error Handling and Troubleshooting in xfNetLink Java

Troubleshooting Techniques

1. Use the `debugInit()` method to initiate a debug session. This method binds an IP address and port number for listening, and then returns the IP and port. You need to include code that displays the IP address (in hex) and port on the screen.

For example:

```
//instantiate objects
AppLogin appLog = new AppLogin();
String clientIP = new String("tiger");
StringBuffer listHost = new StringBuffer("");
StringBuffer listPort = new StringBuffer("");
//make call
appLog.debugInit(clientIP, listHost, listPort);
//display hex IP and port
System.out.println("IP = " + listHost.toString());
System.out.println("Port = " + listPort.toString());
```

2. When the IP and port display on the screen, write them down. You'll need them in [step 4](#). For example:

```
IP = 6F16212C
Port = 1082
```



Once the IP address, etc. displays on the web server screen, you have a limited amount of time in which to manually start *xfServerPlus* in debug mode on the server machine, specify a breakpoint, and type "go". This time is controlled by the `xf_DebugSessionConnectTimeout` setting in the *xfNetLink Java* properties file (see ["Connect session time-out" on page 10-9](#)). If no time-out is specified, the default value of 10 minutes applies. If you delay longer than the time-out value, `SWPConnect` will time out while waiting for a response from *xfServerPlus*.

3. Use `debugStart()` to complete the connection process:

```
appLog.debugStart(clientIP);
```

At this point, the client application has opened a socket and is waiting for the server to call it back.

4. Go to the machine running *xfServerPlus*, start **xfpl.dbr**, and pass the hexadecimal IP address and port to *xfServerPlus*. Type the alpha characters in the IP address in uppercase.

```
dbr -d xfpl hexadecimal_ip listen_port
```

For example:

```
dbr -d xfpl 6F16212C 1082
```

xfServerPlus starts up in the Synergy debugger.

5. Set an initial breakpoint in the **xfpl** program at the `XFPL_DEBUG_BREAK` routine. In the debugger, enter

```
break xfpl_debug_break
```

and then enter

```
go
```

xfServerPlus is now connected to the client on the specified port. The server waits while the client program resumes and makes its first call. The program will then break at the `XFPL_DEBUG_BREAK` routine. This breakpoint occurs just after *xfServerPlus* has opened the ELB for the first method called by your application. (Note that any ELBs linked to this ELB will also be opened.) The ELB must be opened before you can set breakpoints in the routines within it.

6. If the Synergy routine you need to debug is in one of the opened ELBs, just specify a breakpoint in that routine. If the routine you want to debug is in a different (unopened) ELB, use the `OPENELB` debugger command to open that ELB. (You can also continue running your client application until the ELB is opened by *xfServerPlus*. However, because you set a breakpoint at `XFPL_DEBUG_BREAK`, the program will break at each method call, so using the `OPENELB` command is more efficient.)



For general information about the Synergy debugger, see the [“Debugging Your Synergy Programs”](#) chapter in *Synergy Language Tools*. For details on the `OPENELB` command, see [OPENELB](#) in that same chapter.

Running in debug mode on OpenVMS

Use this procedure if the operating system of your *xfServerPlus* machine is OpenVMS.

1. Make sure *xfServerPlus* is running on an unused port. If necessary, restart it to ensure that it's using an unused port.
2. On the machine running *xfServerPlus*, enter

```
$ run DBLDIR:xfpld
```

You'll see output similar to the following:

```
*****
***  DEBUG 9.3.1  ***
BREAK AT 152 IN XFPL (LAUNCHER.DBL;6) ON ENTRY
%DBG-E-Could not open source file "LAUNCHER.DBL;1"
Dbldb>
*****
```



If you have created shared image logicals for the shared images used by *xfServerPlus*, you can skip [step 3](#). Instead, set a breakpoint for your shared image and routine as described in [step 6](#). You'll then be prompted for the port number ([step 4](#)). Once you start your client program ([step 5](#)), the debug session will break at the breakpoint you set.

3. Set an initial breakpoint in the *xfpl* program at the XFPL_DEBUG_BREAK routine. In the debugger, enter

```
set break xfpl_debug_break
```


and then enter

```
go
```
4. When prompted, enter the port number that *xfServerPlus* is running on (from [step 1](#)).
5. Start your client application in the usual manner. After *xfNetLink* connects, the debug session will break at the XFPL_DEBUG_BREAK routine.
6. Set a breakpoint for your Synergy shared image and routine:

```
set break image/routine
```

(For details, see [BREAK](#) in the “Debugging Your Synergy Programs” chapter of *Synergy Language Tools*.)

Note that if you set a breakpoint at XFPL_DEBUG_BREAK, the debugger will break at XFPL_DEBUG_BREAK for each method call your client makes.



Although you do not need to use the OPENELB debugger command before setting the first breakpoint in your shared image, you may need to use it if your code does an XSUBR or RCB_SETFNC without specifying a shared image. For details on the OPENELB command, see [OPENELB](#) in the “Debugging Your Synergy Programs” chapter of *Synergy Language Tools*.

xfNetLink Java Class Reference

This chapter is a reference guide to the Java classes included with xfNetLink Java Edition. If you are generating Java class wrappers, you do not need to use these classes directly.

Synergex.net package

SynProxyNetException 12-3
xfServerNackException 12-4
xfServerPlusUnavailableException 12-6

Synergex.synProxy package

BadSynPacketException 12-7
SynergyWebProxy 12-8
SynProxyBadPropsException 12-17
SynProxyException 12-18
SynProxyIncompatibleServerException 12-19
xfLaunchFatalException 12-20
xfRemoteExecException 12-22

Synergex.synTypes package

ArrayDescriptor 12-24
FieldDescriptor 12-25
ImpDecDescriptor 12-26
IStructManager 12-27
ScalarDescriptor 12-28
StructDescriptor 12-29
synAlpha 12-30
synArray 12-32
synArrayList 12-36

xfNetLink Java Class Reference

synByteArray 12-37
synDec 12-39
SynergyType 12-42
synHandle 12-44
synImpDec 12-46
synNull 12-48
SynProxyBadTypeException 12-49
SynProxyInvalidStructException 12-50
SynProxyMixedTypeArrayException 12-51
synStruct 12-52

Synergex.util package

SWPConnect 12-53
SWPManager 12-55
xfJCWException 12-57
xfPoolException 12-58

Synergex.net.SynProxyNetException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.net.SynProxyNetException

public class SynProxyNetException extends SynProxyException

Signals that the link between the SynergyWebProxy and *xfServerPlus*'s remote execution logic did not occur correctly. Does not include I/O exceptions such as socket failure.

This exception can indicate any of the following situations:

- ▶ The version of *xfServer* that is running does not support *xfServerPlus*. (Windows, UNIX)
- ▶ *xfNetLink* timed out before an *xfServerPlus* remote session was created.
- ▶ *xfServerPlus* was started without the `-w` option. For more information on starting *xfServerPlus*, see [“Running xfServerPlus” on page 3-3](#). (Windows, UNIX)

For more information on errors, see [“Handling Exceptions” on page 11-1](#).

Synergex.net.xfServerNackException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.net.xfServerNackException

public class xfServerNackException extends SynProxyException

Signals that *xfServerPlus* cannot start a remote execution session due to one of the following:

- ▶ *xfServerPlus* is not licensed.
- ▶ The number of *xfServerPlus* license connections has been exceeded (OpenVMS; for other platforms, see “[xfServerPlusUnavailableException](#)” on [page 12-6](#)).
- ▶ *xfServerPlus* cannot access the license database.
- ▶ The *xfNetLink* client version is incompatible with the *xfServerPlus* version.
- ▶ There is an invalid user name or password, or some other problem with the user account used to start *xfServerPlus*.

Generates error message “*xfServer* denied access to remote execution.” The error message includes an operating system error number, the operating system type, and an *xfServerPlus* status code; see the table on [page 3-16](#) for the meaning of the status code.

You can use the `getMessage()` method to obtain the standard error message or use the methods below to create your own message.

For more information on errors, see “[Handling Exceptions](#)” on [page 11-1](#).

Methods public String `getMessage()`

Overrides `Exception.getMessage()` method in class `Throwable` to create a formatted error message string that can be reported to the user. To create your own format for the message, use the accessor methods listed below for each of the attributes (e.g., system error number) stored in this class.

public int `getSystemErr()`

Returns the operating system error number that is set when *xfServerPlus* is denied service.

```
public String getSystemTypeAsString()
```

Returns, as a string, the type of operating system on the machine where *xfServerPlus* is running.

```
public int getSystemType()
```

Returns a numeric code indicating the operating system on the machine where *xfServerPlus* is running:

1 = UNIX

2 = OpenVMS

3 = Windows

Use this method to get the required input parameter for `getxfStatusAsString()`.

```
public int getxfStatus()
```

Returns the *xfServerPlus* status code that is set when service is denied; see the table on [page 3-16](#). Use this method to get the required input parameter for `getxfStatusAsString()`.

```
public String getxfStatusAsString(int os, int xfStatus)
```

Returns the *xfServerPlus* status message that is set when service is denied. For example:

```
String explanation = new String(getxfStatusAsString  
(getSystemType(), getxfStatus()));
```

os—the numeric code for type of operating system. See list above.

xfStatus—the numeric code for *xfServerPlus* error status; see the table on [page 3-16](#).

```
public boolean invalidLogin()
```

Returns true or false to indicate whether the *xfServerPlus* status message indicated an invalid password or user name as the reason for denial of service.

Synergex.net.xfServerPlusUnavailableException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.net.xfServerPlusUnavailableException

public class xfServerPlusUnavailableException extends
SynProxyException

Signals that *xfServerPlus* did not start successfully. This exception can indicate any of the following situations:

- ▶ The 14-day demo period or an extended demo period has expired (Windows, UNIX).
- ▶ The number of licensed connections has been exceeded (Windows and UNIX; for OpenVMS see “[xfServerNackException](#)” on page 12-4).
- ▶ A serious error, such as inadequate memory, occurred while *xfServerPlus* was starting.

This exception generates the following error messages:

- ▶ 14-day demo period expired.
- ▶ Extended demo period expired.
- ▶ Number of licensed concurrent connections exceeded.
- ▶ Severe system error.

For more information on errors, see “[Handling Exceptions](#)” on page 11-1.

Synergex.synProxy.BadSynPacketException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.synProxy.BadSynPacketException

public class BadSynPacketException extends SynProxyException

Signals that the message being sent to or received from *xyServerPlus* is formatted incorrectly. This exception can indicate either of the following situations:

- ▶ The packet is corrupted or formatted incorrectly.
- ▶ The method ID is missing.

Under normal circumstances, you should not see this exception. For more information on errors, see [“Handling Exceptions” on page 11-1](#).

Synergex.synProxy.SynergyWebProxy

java.lang.Object

└─ Synergex.synProxy.SynergyWebProxy

public class SynergyWebProxy extends Object

The `SynergyWebProxy` class is the interface between a Java application and a Synergy routine. The `SynergyWebProxy.xcall()` acts like a Synergy Language XCALL statement.

Constructors public `SynergyWebProxy(String host, int port, InetAddress client, int minport, int maxport, boolean debugtrace, String debugtracefile)` throws `UnknownHostException`, `IOException`, `xfServerPlusUnavailableException`, `SynProxyNetException`, `xfServerNackException`



The port number range feature will be deprecated in a future version, which means this constructor will be deprecated also. We recommend that you use a constructor that does not pass *minport* and *maxport*.

Connects to the specified host name on the specified port. With this constructor you can specify a port number range for `SynergyWebProxy` to use and turn on debug trace output without using an `xfNetLink` Java properties file. For additional information on the port range and debug output, see “[Configuring the xfNetLink Java Properties File](#)” on page 10-2 and “[Client-Side Logging](#)” on page 11-7.

host—the host name of the machine where `xfServerPlus` is running

port—the port number that `xfServerPlus` is listening on for remote session requests (default is 2356)

client—the `InetAddress` of the client (`xfNetLink`) machine

minport—the beginning value for the port number range that `SynergyWebProxy` uses to communicate with `xfServerPlus`

maxport—the ending value for the port number range that `SynergyWebProxy` uses to communicate with `xfServerPlus`

`debugtrace`—boolean specifying whether to display debug output (packets sent and received)

`debugtracefile`—filename to which the debug output will be written. You can include the full path name. If you want the output sent to the screen instead of to a file, set this to 'null'. If *debugtrace* is set to 'false', set this to 'null'.

```
public SynergyWebProxy(InetAddress netHost, int port,  
    InetAddress client) throws UnknownHostException, IOException,  
    xfServerPlusUnavailableException, SynProxyNetException,  
    xfServerNackException
```

Connects to the specified host address on the specified port.

`netHost`—the `InetAddress` of the machine where *xfServerPlus* is running

`port`—the port number that *xfServerPlus* is listening on for remote session requests

`client`—the `InetAddress` of the client (*xfNetLink*) machine

```
public SynergyWebProxy(String host, int port,  
    InetAddress client) throws UnknownHostException, IOException,  
    xfServerPlusUnavailableException, SynProxyNetException,  
    xfServerNackException
```

Connects to the specified host name on the specified port.

`host`—the host name of the machine where *xfServerPlus* is running

`port`—the port number that *xfServerPlus* is listening on for remote session requests

`client`—the `InetAddress` of the client (*xfNetLink*) machine



Any constructor that checks the properties file for host name and/or port, also checks that file for the `SynergyWebProxy` port number range and the debug output settings. If you're using a properties file, the port number range and debug output are optional. See ["Configuring the xfNetLink Java Properties File" on page 10-2](#) and ["Client-Side Logging" on page 11-7](#) for more information.

xfNetLink Java Class Reference

SynergyWebProxy

```
public SynergyWebProxy(InetAddress client) throws  
    UnknownHostException, IOException,  
    xfServerPlusUnavailableException, SynProxyNetException,  
    xfServerNackException, SynProxyBadPropsException
```

Connects to the host on the port defined in the *xfNetLink* Java properties file. This constructor assumes that the properties file is named **xfNetLnk.ini** and is located in the directory that the application containing the SynergyWebProxy was started in.

client—the `InetAddress` of the client (*xfNetLink*) machine

```
public SynergyWebProxy(InetAddress client, File propFile) throws  
    UnknownHostException, IOException,  
    xfServerPlusUnavailableException, SynProxyNetException,  
    xfServerNackException, SynProxyBadPropsException
```

Connects to the host on the port defined in the specified properties file.

client—the `InetAddress` of the client (*xfNetLink*) machine

propFile—the `File` object that names the file that contains the settings for `xf_RemoteHostName` and `xf_RemotePort`

```
public SynergyWebProxy(String host, InetAddress client,  
    File propFile) throws UnknownHostException, IOException,  
    xfServerPlusUnavailableException, SynProxyNetException,  
    xfServerNackException, SynProxyBadPropsException
```

Connects to the specified host on the port defined in the specified *xfNetLink* Java properties file.

host—the name of the host machine where *xfServerPlus* is running

client—the `InetAddress` of the client (*xfNetLink*) machine

propFile—the `File` object that names the file that contains the setting for `xf_RemotePort`

```
public SynergyWebProxy(int port, InetAddress client,  
    File propFile) throws UnknownHostException, IOException,  
    xfServerPlusUnavailableException, SynProxyNetException,  
    xfServerNackException, SynProxyBadPropsException
```

Connects, on the specified port, to the host defined in the specified *xfNetLink* Java properties file.

port—the port number that *xfServerPlus* is listening on for remote session requests

client—the `InetAddress` of the client (*xfNetLink*) machine

propFile—the `File` object that names the file that contains the setting for `xf_RemoteHostName`

```
public SynergyWebProxy(InetAddress netHost, InetAddress client,  
    File propFile) throws UnknownHostException, IOException,  
    xfServerPlusUnavailableException, SynProxyNetException,  
    xfServerNackException, SynProxyBadPropsException
```

Connects to the specified host on the port defined in the specified *xfNetLink* Java properties file.

netHost—the `InetAddress` of the machine where *xfServerPlus* is running

client—the `InetAddress` of the client (*xfNetLink*) machine

propFile—the `File` object that names the file that contains the setting for `xf_RemotePort`

```
public SynergyWebProxy(InetAddress client, boolean debug) throws  
    IOException, SynProxyException, SynProxyBadPropsException
```

Allows a remote session to be launched for debugging purposes. To debug Synergy code in the ELBs or shared images that are being called from *xfServerPlus*, instantiate a `SynergyWebProxy` using this constructor and set *debug* to 'true'. The instantiation will fail if *debug* is set to 'false'; if you don't want to run in debug mode, use a different constructor. This constructor is not needed for debug mode if your server is on OpenVMS. See [“Running an xfServerPlus Session in Debug Mode” on page 11-11](#) for more information.

If you are using a properties file *and* it is named `xfNetLnk.ini` *and* it is located in the directory that the Java application containing the `SynergyWebProxy` was started in, this constructor will check that file for the `SynergyWebProxy` port number range, debug output settings, and a connection time-out value. See [“Configuring the xfNetLink Java Properties File” on page 10-2](#) for additional information.

client—the `InetAddress` of the client (*xfNetLink*) machine

debug—boolean value to distinguish this constructor from the non-debug version. Set this to true to run the `SynergyWebProxy` in debug mode. You can also use the mnemonic `SynergyWebProxy.DO_DEBUG` for this.

```
public SynergyWebProxy(InetAddress client, boolean debug,  
    int connectTimeout) throws IOException, SynProxyException,  
    SynProxyBadPropsException
```

Allows a remote session to be launched for debugging purposes. This constructor is the same as the debug constructor above, except that it allows you to pass a connection time-out value in the constructor rather than using the value in the properties file or the default value. This constructor still checks the **xfNetLnk.ini** file for the port number range and the debug output settings as described above.

This constructor is not needed for debug mode if your server is on OpenVMS. See [“Running an xfServerPlus Session in Debug Mode” on page 11-11](#) and [“Configuring the xfNetLink Java Properties File” on page 10-2](#) for additional information.

client—the `InetAddress` of the client (*xfNetLink*) machine

debug—boolean value to distinguish this constructor from the non-debug version. Set this to true to run the `SynergyWebProxy` in debug mode. You can also use the mnemonic `SynergyWebProxy.DO_DEBUG` for this.

connectTimeout—the amount of time, in seconds, that *xfNetLink* will wait for a startup acknowledgment from *xfServerPlus*. This is ‘B’ in [figure 5-1 on page 5-4](#). The time-out value specified in this constructor overrides the value specified for `xf_DebugSessionConnectTimeout` in the properties file. For more information about time-outs, see [“Specifying Time-out Values” on page 10-8](#).

```
public SynergyWebProxy(InetAddress client, File propFile,  
    boolean debug) throws IOException, SynProxyException,  
    SynProxyBadPropsException
```

Allows a remote session to be launched for debugging purposes. This constructor is the same as the two debug constructors above, except that it allows you to specify the properties file that contains the port number range for `SynergyWebProxy` and the settings for debug output and connection time-out. This constructor is not needed for debug mode if your server is on OpenVMS.

See [“Configuring the xfNetLink Java Properties File” on page 10-2](#), [“Client-Side Logging” on page 11-7](#), and [“Running an xfServerPlus Session in Debug Mode” on page 11-11](#) for more information.

client—the `InetAddress` of the client (*xfNetLink*) machine

propFile—the `File` object that names the file that contains settings for the `SynergyWebProxy` port range and debug output

debug—boolean value to distinguish this constructor from the non-debug version. Set this to true to run the `SynergyWebProxy` in debug mode. You can also use the mnemonic `SynergyWebProxy.DO_DEBUG` for this.

`public SynergyWebProxy(InetAddress client, boolean debug, String anyString)` throws `IOException`, `SynProxyException`, `SynProxyBadPropsException`

Initiates a remote session for debugging purposes. You can use this constructor with the `SWP_Start()` method instead of the above debug constructors.

You must include code to display the client IP address and port in hex. Use the `getlisten_ip()` and `getlisten_port()` methods to retrieve them. Use the `SWP_Start()` method to complete the connection process.

client—the `InetAddress` of the client (*xfNetLink*) machine

debug—boolean value to indicate this is a debug constructor. Set this to true to run the `SynergyWebProxy` in debug mode.

anyString—Any string value. This parameter is required to make the constructor unique. It is not used, but it *must* be passed.

Methods `public SynergyType xcall(String methId, [SynergyType arg, ..., SynergyType arg20])` throws `IOException`, `SynProxyException`, `xfRemoteExecException`, `xfLaunchFatalException`

Acts like a Synergy Language XCALL statement. This method formats the routine and up to 20 parameters to be sent to *xfServerPlus*. It returns the value returned from the remote `xcall()` as a `SynergyType` object, which should be cast (converted) to the appropriate `SynergyType` subclass. Return values are placed in the parameters.

methId—a string containing the method ID of the routine to call

arg1...arg20—(optional) up to 20 `SynergyType` parameters

`public SynergyType xcall(String methId, SynergyType[] argv)` throws `IOException`, `SynProxyException`, `xfRemoteExecException`, `xfLaunchFatalException`

Acts like a Synergy Language XCALL statement. This method enables you to pass up to 253 parameters in an array; use it when you need to pass more than 20 parameters. It returns the value returned from the remote `xcall()` as a `SynergyType` object, which should be cast (converted) to the appropriate

SynergyType subclass. Return values are placed in the parameters. See [“To make a call with more than 20 parameters” on page 12-16](#) for instructions on using this method.

methId—a string containing the method ID of the routine to call

argv—a SynergyType array containing up to 253 elements

```
public void sendShutdownMessage() throws IOException,  
    SynProxyException
```

Sends a message to *xfServerPlus* to close the connection.

```
public int resetRemoteCallTimeout(int seconds)
```

Resets the call time-out value. This time-out is measured for each send–receive request between *xfNetLink* and *xfServerPlus*. For more information about time-outs, see [“Specifying Time-out Values” on page 10-8](#).

seconds—the amount of time, in seconds, that *xfNetLink* should wait for a return from a remote call. The time-out value specified in this method overrides the value for *xf_SessionLingerTimeout* in the properties file.

```
public int getRemoteCallTimeout()
```

Returns the current call time-out value. This method get the time-out value regardless of how it was set: it may be the default value, the value in the *xfNetLink* Java properties file, or the value set with *resetRemoteCallTimeout*. For more information about time-outs, see [“Specifying Time-out Values” on page 10-8](#).

```
public int SWP_Start(InetAddress client, boolean debug) throws  
    IOException, SynProxyException, SynProxyBadPropsException
```

Completes the connection process started by the debug constructor *SynergyWebProxy(InetAddress client, boolean debug, String anyString)*.

client—the IP address or name of the client (*xfNetLink*) machine

debug—boolean value to indicate this is a debug method. Set this to true to run the *SynergyWebProxy* in debug mode.

Usage The `SynergyWebProxy.xcall()` method acts like a Synergy Language XCALL statement: it calls a Synergy routine and passes parameter data. The routine must have an entry in the Synergy Method Catalog.

To use the `SynergyWebProxy.xcall()` method, pass the following as parameters:

- ▶ The method ID (as entered in the SMC) of the routine that you want to call
- ▶ Any parameter data, packaged in `SynergyType` objects

Using the standard `xcall()` method, you can pass a maximum of 20 parameters in a single call. If you need to pass more than 20 parameters, use the `xcall()` method that allows you to pass an array of up to 253 parameters. (See [“To make a call with more than 20 parameters” on page 12-16.](#))

If there are return values, they are passed back in the parameters or as a result of the Synergy function. (See [“To call a Synergy function with a return value” on page 12-15.](#))



The method ID is case sensitive. The method ID in the `xcall()` must be exactly the same as it was entered in the SMC.

▶ To call a Synergy routine *without* a return value

To call a subroutine or a function for which you do not want to catch the return value, do this:

```
swp1.xcall("methodid1234", arg1, arg2, arg3);
```

where *methodid1234* is the method ID in the SMC and *arg1*, *arg2*, *arg3* are `SynergyType` parameters that you are passing.

▶ To call a Synergy function *with* a return value

If the Synergy routine that you are calling is a function and you want to catch the return value, you need to determine the type of data you are catching and create a `SynergyType` variable to catch it.

For example:

```
//declare variables
synDec d1;           //data that you are passing
synAlpha a1;         //data that you are passing
synDec returnValue; //this will catch the returned data

//make the xcall and cast the return value as a synDec
returnValue = (synDec)swp1.xcall("methodid1234", d1, a1);
```

► To make a call with more than 20 parameters

You can pass up to 253 parameters by using the `xcall()` method that allows you to pass a `SynergyType` array.

1. Declare the `SynergyTypes` that are going to be elements in the array. You can combine different types of `SynergyTypes` in the array, including `SynArrays`, if desired. For example:

```
synAlpha a1 = new synAlpha ("Bruce Foster");
synDec d1 = new synDec (123456);
synArray array1 = new synArray(3);

//assign values for the synArray
array1.array[0] = new synDec(111);
array1.array[1] = new synDec(222);
array1.array[2] = new synDec(333);
```

2. Declare the `SynergyType` array:
3. Instantiate a new `SynergyType` array with three elements:

```
paramArray = new SynergyType[3];
```

4. Assign values to the array elements:

```
paramArray[0] = a1;
paramArray[1] = d1;
paramArray[2] = array1;
```

5. Make the call, passing the method ID and the array:

```
swp1.xcall("methodID", paramArray);
```


Synergex.synProxy.SynProxyBadPropsException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.synProxy.SynProxyBadPropsException

public class SynProxyBadPropsException extends SynProxyException

Signals that the *xfNetLink* Java properties file contains erroneous or missing information. This exception can indicate any of the following situations:

- ▶ The value specified for `xf_RemotePort` or `xf_RemoteHostName` is null.
- ▶ The value specified for `xf_RemotePort`, `xf_LocalMinPort`, or `xf_LocalMaxPort` is not an integer.
- ▶ The value specified for `xf_LocalMinPort` or `xf_LocalMaxPort` is less than or equal to 1024.
- ▶ The value specified for `xf_LocalMaxPort` is less than the value specified for `xf_LocalMinPort`.

For more information on the properties file, see [“Configuring the xfNetLink Java Properties File” on page 10-2](#). For more information on errors, see [“Handling Exceptions” on page 11-1](#).

Synergex.synProxy.SynProxyException

`java.lang.Exception`

└─ Synergex.synProxy.SynProxyException

`public class SynProxyException extends Exception`

The abstract class from which all `SynProxyException` subclasses are derived. This class is exactly the same as the `java.lang.Exception` class from which it is derived.

Synergex.synProxy.SynProxyIncompatibleServerException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.synProxy.
SynProxyIncompatibleServerException

```
public class SynProxyIncompatibleServerException extends  
SynProxyException
```

Signals that the version of *xfServerPlus* is not compatible with the version of the client for the current operation.

Generates error message “Server version *n* is incompatible with the required version *n*” where *n* is the protocol version of the server. Refer to the table below for the corresponding software version. The message also includes the routine that was being called when the error occurred and the number of the parameter that caused the error. You can use the `getMessage()` method to obtain the standard error message or use the methods below to create your own message.

Protocol version	xfServerPlus version
2	6.3, 7.1
3	7.3
4	7.5
5	8.1, 8.3, 9.1
6	9.3

Methods `public int getParamNumber()`

Returns the number of the parameter that caused the error.

`public int getRequiredVersion()`

Returns the required protocol version for the aborted operation.

`public int getServerVersion()`

Returns the protocol version of the *xfServerPlus* server.

`public String getSubroutineName()`

Returns the name of the routine (method) that was being called when the error occurred.

Synergex.synProxy.xfLaunchFatalException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.synProxy.xfLaunchFatalException

public class xfLaunchFatalException extends SynProxyException

Signals that *xfServerPlus* encountered a fatal error and aborted. This exception generally indicates that an untrapped fatal error occurred in a Synergy routine that *xfServerPlus* was executing. When this exception is thrown, the *xfServerPlus* remote session has already quit running.

Generates error message “Remote Synergy execution aborted.” The message also includes supplemental error text, program name, routine name, the line number at which the error occurred, and the operating system error that was set at the time of failure.

You can use the `getMessage()` method to obtain the standard error message or use the methods below to create your own message.

For more information on errors, see [“Handling Exceptions” on page 11-1](#).

Methods public String `getProgramName()`

Returns the name of the failed program running on the remote server. For Synergy programs, this is the name defined in the MAIN statement or the name of the `dbf` file if there is no name defined in MAIN.

public String `getSubroutineName()`

Returns the name of the routine running on the remote server in which the fatal error occurred.

public int `getErrorAtLineNumber()`

Returns the line number in the Synergy program file at which the fatal error occurred.

public String `getErrorAtLineNumberAsString()`

Returns a string containing the line number in the Synergy program file that was executing when the fatal error occurred.

```
public int getLastRemoteSystemErrNo()
```

Returns the last operating system level error on the system when the fatal error was trapped.

```
public String getLastRemoteSystemErrNoAsString()
```

Returns a string containing the last operating system level error on the system when the fatal error was trapped.

```
public String getSupplementalErrorText()
```

Returns any explanatory information posted by the failing program.

```
public String getOsType()
```

Returns the category of operating system that *xfServerPlus* is running on (Windows, UNIX, or OpenVMS).

Synergex.synProxy.xfRemoteExecException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.synProxy.xfRemoteExecException

public class xfRemoteExecException extends SynProxyException

Signals that the executing routine terminated or was not called due to problems detected while setting up the call. *xfServerPlus* does not terminate when this exception occurs.

Generates error message “Failure occurred within remote Synergy routine.” The message also includes the method ID, the error number, whether it’s an *xfServerPlus* or method error, the error message text, and the supplemental error text.

You can use the `getMessage()` method to obtain the standard error message or use the methods below to create your own message.

For more information on errors, see [“Handling Exceptions” on page 11-1](#).

Methods public String `getMethodId()`

Returns the method ID that was passed to *xfServerPlus*. The method ID identifies a Synergy routine and the ELB or shared image in which it is located. To get the actual routine name, you may need to refer to the Synergy Method Catalog (see [“Searching for Methods and Parameters” on page 2-37](#)).

public String `getErrorNumber()`

Returns the error number returned by *xfServerPlus* or by the routine in which the error occurred.

public String `getErrorMessage()`

Returns the error message and supplemental information returned by *xfServerPlus* or by the routine in which the error occurred.

public boolean `errorDetectedByxfServerPlus()`

Returns true if the error is an *xfServerPlus* error.

public boolean `errorIsInSynergyMethod()`

Returns true if the error is in the Synergy code in the ELB or shared image.

```
public char getErrorType()
```

Returns the error type: L = an error in *xfServerPlus*; M = an error in the ELB or shared image.

Synergex.synTypes.ArrayDescriptor

java.lang.Object

└─ Synergex.synTypes.FieldDescriptor

└─ Synergex.synTypes.ArrayDescriptor

public class ArrayDescriptor extends FieldDescriptor

Describes a structure field that is an array.

The `ArrayDescriptor` class is used by the Java code that is created when you generate Java class wrappers. You do not need to call this class directly.

Synergex.synTypes.FieldDescriptor

`java.lang.Object`

└─ Synergex.synTypes.FieldDescriptor

`public abstract class FieldDescriptor extends java.lang.Object`

Describes a field within a structure. This is the superclass of available `FieldDescriptor` classes.

The `FieldDescriptor` class is used by the Java code that is created when you generate Java class wrappers. You do not need to call this class directly.

Synergex.synTypes.ImpDecDescriptor

`java.lang.Object`

└─ `Synergex.synTypes.FieldDescriptor`

└─ `Synergex.synTypes.ScalarDescriptor`

└─ `Synergex.synTypes.ImpDecDescriptor`

`public class ImpDecDescriptor extends ScalarDescriptor`

Describes an implied-decimal field within a structure. An implied-decimal field is a special type of scalar field because it has a precision value in addition to its field size.

The `ImpDecDescriptor` class is used by the Java code that is created when you generate Java class wrappers. You do not need to call this class directly.

Synergex.synTypes.IStructManager

```
public interface IStructManager
```

The `IStructManager` interface is a set of methods implemented by any class that is used to represent a structure data type. This interface enables the `synStruct SynergyType` to query the class for its structure and field information and then build the necessary descriptors.

The `IStructManager` interface is used by the Java code that is created when you generate Java class wrappers. You do not need to implement this interface directly.

Synergex.synTypes.ScalarDescriptor

java.lang.Object

└─ Synergex.synTypes.FieldDescriptor

└─ Synergex.synTypes.ScalarDescriptor

```
public class ScalarDescriptor extends FieldDescriptor
```

Describes a scalar field within a structure. The scalar types are alpha, decimal, implied-decimal, and integer. (Implied-decimal is a special case, however, and is handled by its own subclass; see [page 12-26](#).)

The `ScalarDescriptor` class is used by the Java code that is created when you generate Java class wrappers. You do not need to call this class directly.

Synergex.synTypes.StructDescriptor

`java.lang.Object`

└─ `Synergex.synTypes.FieldDescriptor`

└─ `Synergex.synTypes.StructDescriptor`

`public class StructDescriptor extends FieldDescriptor`

Describes a Synergy structure. A structure is made up of a number of fields, each of which is described by a `FieldDescriptor`.

The `StructDescriptor` class is used by the Java code that is created when you generate Java class wrappers. You do not need to call this class directly.

Synergex.synTypes.synAlpha

java.lang.Object

└─ Synergex.synTypes.SynergyType

└─ Synergex.synTypes.synAlpha

public class synAlpha extends SynergyType

A wrapper for a Java String; corresponds to Synergy's alpha data type.

Constructors public synAlpha(String in)

Constructs a synAlpha object representing the string.

in—the string to be stored

public synAlpha()

Constructs a synAlpha object with a blank string. Use this when you want to return results, but do not want to send data.

Methods public String value()

Returns the string stored in this object.

public void value(String in)

Sets the string stored in this object.

in—the string to be stored

public String type()

Returns a string ("AL") that describes the SynergyType object.

Overrides type() in class SynergyType.

Usage Use the synAlpha class to translate a Java String into a Synergy alpha data type.
For example:

```
public String getAlphaData() throws RemoteException
{
    //declare variables
    synAlpha    a1,a2,a3;
    String      var1 = "Paula Strand";
    String      var2 = "566-77-8888";
    String      var3 = "01-12-56";
```

```
//to catch return values
String      returnData1, returnData2, returnData3;
.
.
.
//instantiate new synAlpha objects to package Java variables
a1 = new synAlpha(var1);
a2 = new synAlpha(var2);
a3 = new synAlpha(var3);

//make the xcall and pass the parameters
swp.xcall("methodid0004", a1, a2, a3);

//extract the returned data
returnData1 = a1.value();
returnData2 = a2.value();
returnData3 = a3.value();
.
.
.
}
```

To retrieve a string from a synAlpha object, invoke the value() method. For example:

```
String returnData = a1.value();
```

Synergex.synTypes.synArray

java.lang.Object

└─ Synergex.synTypes.SynergyType

└─ Synergex.synTypes.synArray

public class synArray extends SynergyType

Implements a one-dimensional array.

Variables public SynergyType array[]

Allows direct access to array elements via synArray.array[i].

Constructors public synArray(int numElements)

Constructs a synArray object containing the specified number of elements. Use this constructor when you are passing an array with specific values to Synergy.

numElements—the number of elements in the array

public synArray(int numElements, synDec init)

Constructs a synArray object containing the specified number of synDec elements initialized to the value passed in the synDec object. Use this constructor to create an array of synDecs that is empty and will be filled by your Synergy routines, or to create an array that will be initialized by Synergy, or when the array is not full.

numElements—the number of elements in the array

init—the object containing the value that you want the synDec array initialized to

public synArray(int numElements, synAlpha init)

Constructs a synArray object containing the specified number of synAlpha elements initialized to the value passed in the synAlpha object. Use this constructor to create an array of synAlphas that is empty and will be filled by your Synergy routine, or to create an array that will be initialized by Synergy, or when the array is not full.

numElements—the number of elements in the array

init—the object containing the value that you want the synAlpha array initialized to


```
public synArray(int numElements, synImpDec init)
```

Constructs a `synArray` object containing the specified number of `synImpDec` elements initialized to the value passed in the `synImpDec` object. Use this constructor to create an array of `synImpDecs` that is empty and will be filled by your Synergy routine, or to create an array that will be initialized by Synergy, or when the array is not full.

`numElements`—the number of elements in the array

`init`—the object containing the value that you want the `synImpDec` array initialized to

Methods `public void assignElement(int pos, SynergyType object)`

Assigns the specified `SynergyType` object to the specified array position. This is a zero-based index.

`pos`—the position in the array that the object should be assigned to

`object`—the object passed to the array

```
public SynergyType getElement(int pos)
```

Returns the array element corresponding to the specified array position. This is a zero-based index.

`pos`—the position of the array element

```
public int length()
```

Returns the number of elements in the array.

```
public String type()
```

Returns a string (“AR”) that describes the `SynergyType` object.

Overrides `type()` in class `SynergyType`.

Usage Use the `synArray` class to pass one-dimensional arrays. The `synArray` object contains a Java attribute named “array” that can be used to access individual array elements.



If you use multi-dimensional arrays or vectors in Java, decompose them into one-dimensional arrays before packaging them in a `synArray` object.

This example illustrates how to create an array that passes values to Synergy, and how to extract the returned values.

```
//declare variables for the array and its elements
```

xfNetLink Java Class Reference

synArray

```
synArray    ar1;
int[]       varArray{31, 53, 9, 100};
.
.
.
//instantiate a synArray object with 4 elements
ar1 = new synArray(4);
//assign objects to elements; include synArray object name (ar1)
// because the array is inside synArray
for (index=0; index<4; index++)
    ar1.array[index] = new synDec(varArray[index]);

//make the xcall
swp.xcall("methodid0017", ar1);
//extract the returned data
for (index=0; index<4; index++)
    varArray[index] =
        ((synDec)ar1.array[index]).value();
```

This example illustrates how to create an empty array that can be filled by a Synergy routine.

```
//instantiate a blank synAlpha
synAlpha a1 = new synAlpha();

//instantiate a synArray object with 6 blank synAlpha elements
synArray AlphaArray = new synArray(6, a1);

//make the xcall
swp.xcall("methodid0018", AlphaArray);
```

You can also use the `synArray.assignElement()` method to assign elements in an array. For example, instead of

```
synDec id1 = new synDec(15);
ar1.array[0] = id1;
```

do this:

```
synDec id1 = new synDec(15);
ar1.assignElement(0,id1);
```

You can use the `synArray.getElement()` method to extract data from returned values. For example, instead of

```
long var1 = id1.value();
```

do this:

```
long var1 = ((synDec)ar1.getElement(0)).value();
```

To extract one of the non-default data types, such as an int, with the `getElement()` method, do this:

```
int var3 = (int)((synDec)ar1.getElement(2)).value();
```



Because SynergyType does not have any public methods to access values, you must cast the SynergyType object to the appropriate subclass when retrieving data from an array. For example:
`((synAlpha)ar.array[0]).value();`

Synergex.synTypes.synArrayList

java.lang.Object

└─ Synergex.synTypes.SynergyType

└─ Synergex.synTypes.synArrayList

public class synArrayList extends SynergyType

A wrapper for a Java ArrayList, which is used to return a structure collection parameter (array of structures) from Synergy.

The `synArrayList` class is used by the Java code that is created when you generate Java class wrappers. You do not need to call this class directly. See [“Returning a Collection of Structures” on page 1-14](#) for more information.

Synergex.synTypes.synByteArray

java.lang.Object

└─ Synergex.synTypes.SynergyType

└─ Synergex.synTypes.synByteArray

public class synByteArray extends SynergyType

A wrapper for a Java ArrayList; corresponds to a memory handle in Synergy.

Constructors

public synByteArray(byte[] in)

Constructs a synByteArray object from a byte array.

in—the Java byte array to be stored

public synByteArray(java.util.ArrayList al) throws
SynProxyBadTypeException

Constructs a synByteArray object from an ArrayList.

al—the Java ArrayList containing a byte array as the first element

public synByteArray()

Constructs a synByteArray object containing no data. Use this when you want to return results, but do not want to send data.

Methods

public byte[] value()

Returns the byte array stored in this object.

public void value(byte[] in)

Sets the byte array in this object.

in—the byte array to be stored

public String type()

Returns a string (“BI”) that describes the SynergyType object.

Overrides type() in class SynergyType.

public int length()

Returns the length of the byte array.

Usage Use the `synByteArray` class to pass binary data such as JPEG files to and from Synergy.

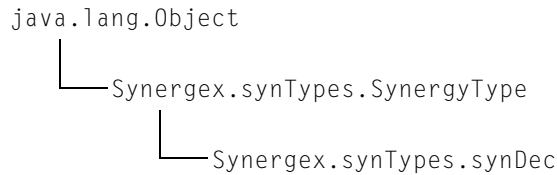
In the SMC, the parameter must be defined as a “Binary (handle)” data type. On the Synergy side, the server routine must declare the argument that receives the data as a memory handle (i4). `xfServerPlus` will place the data in a memory area and pass the memory handle allocated to that area to your Synergy server routine. After the data has been returned to `xfNetLink`, `xfServerPlus` will free the memory area.

The following example shows how to use the `synByteArray` class with an “in/out” parameter. Note that you could also use an `ArrayList`.

```
//instantiate SWP using the xfNetLnk.ini file
SynergyWebProxy SWP = new SWPConnect(m_iniFile);

//create the byte[] and fill it with data
byte[] inba = new byte[67000];
.
.
.
//create synByteArray and pass the byte[] in the constructor
synByteArray synba = new synByteArray(inba);
//call the XFPL method
SWP.xcall("BinaryArrayMethod",synba);
//extract the returned byte array
byte[] rtnba = (byte[])synba.value();
SWP.disconnect();
```

Synergex.synTypes.synDec



`public class synDec extends SynergyType`

A wrapper for Java long, int, byte, and short data types; corresponds to Synergy's decimal and integer data types.

Constructors `public synDec(long in)`
Constructs a synDec object from a long.
`in`—the number to be stored

`public synDec(int in)`
Constructs a synDec object from an int.
`in`—the number to be stored

`public synDec(short in)`
Constructs a synDec object from a short.
`in`—the number to be stored

`public synDec(byte in)`
Constructs a synDec object from a byte.
`in`—the number to be stored

`public synDec()`
Constructs a synDec object containing no data. Use this when you want to return results, but do not want to send data.

Methods `public long value()`
Returns the number stored in this object.

`public void value(long in)`
Sets the number stored in this object. Use when passing a long.
`in`—the number to be passed

```
public void value(int in)
```

Sets the number stored in this object. Use when passing an int.

in—the number to be passed

```
public void value(short in)
```

Sets the number stored in this object. Use when passing a short.

in—the number to be passed

```
public void value(byte in)
```

Sets the number stored in this object. Use when passing a byte.

in—the number to be passed

```
public String type()
```

Returns a string (“DE”) that describes the SynergyType object.

Overrides type() in class SynergyType.

Usage If the value stored in a synDec object is greater than 4,294,967,296 (2^{32}), append an ‘L’ to the end of the value. For example:

```
synDec x = new synDec(5555555555L)
```

Use the synDec class to translate Java data types byte, short, int, and long into Synergy integers and decimals. For example:

```
//pass in three variables
public long myMethod(long var1, byte var2, int var3)
{
    //declare variables
    synDec    d1,d2,d3;

    //instantiate new synDec objects to package Java variables
    d1 = new synDec(var1);
    d2 = new synDec(var2);
    d3 = new synDec(var3);

    //make the xcall and pass the parameters
    swp.xcall("methodid0024", d1, d2, d3);
    //extract the returned data
    var1 = d1.value();
    var2 = (byte)d2.value(); //cast return value to byte
    var3 = (int)d2.value();  //cast return value to int
    return var1 + var2 + var3;
}
```


To retrieve a long from a `synDec` object, invoke the `value()` method. For example:

```
var1 = d1.value();
```

To retrieve a byte, short, or int from a `synDec` object, you must cast the variable to the desired Java data type. For example:

```
var1 = (float)d1.value();
```

Synergex.synTypes.SynergyType

java.lang.Object



public abstract class SynergyType extends Object

The superclass of the available SynergyType classes that can be used in Java. SynergyType wraps Java types so that variable information can be sent to *xyServerPlus* and updated with returned information.

Constructor public SynergyType()

Usage Data must be packaged into SynergyType objects to be passed from Java to Synergy. There are a number of SynergyType subclasses to handle different types of data (alpha, decimal, etc.), as well as a subclass to pass null values. After deciding which SynergyType subclass you need to use, you will instantiate a SynergyType object with the variable data that you want to pass.

The table on [page 12-43](#) shows the correspondence between Synergy Language data types, SynergyType objects, and Java data types. This table is designed from the perspective of the programmer who is working with existing Synergy routines and needs to know which SynergyType objects and Java data types to use.



Because the `synArrayList` and `synStruct` subclasses are not used directly, they are not included with the other SynergyType objects in the table. These two classes are used only by the code that's created when you generate Java class wrappers.

In the first column find the Synergy Language data type that your routine requires. Work backwards from there to determine the SynergyType object and the Java data type you should use. For example, if your Synergy routine is expecting an implied-decimal data type, you should package a `synImpDec` object with a Java variable of type float or double.

Once you have determined the correct SynergyType object and Java data type to use, declare the variables and instantiate a SynergyType object.

For this Synergy Language data type	Package this SynergyType object	With this Java data type
alpha	synAlpha	String
integer (i1, i2, i4, i8)	synDec ^a	byte, short, int, long
decimal	synDec ^a	byte, short, int, long
implied-decimal	synImpDec ^b	float, double
array of alpha elements	synArray of synAlpha	String
array of integer elements	synArray of synDec	byte, short, int, long
array of decimal elements	synArray of synDec	byte, short, int, long
array of implied-decimal elements	synArray of synImpDec	float, double
memory handle	synHandle synByteArray ^c	String ArrayList, byte array

- a. A synDec object can be translated to any Synergy Language data type. See note below.
- b. A synImpDec object can be translated to any Synergy Language data type; however, data will be truncated if it is translated to an integer or decimal. See note below.
- c. Use a synByteArray object when passing binary data.



For best results, choose the Java data type and SynergyType object that correspond most closely to the final Synergy Language data type. Although xfNetLink Java can convert to any data type, return data of the wrong type can cause an exception. For more information on data translation, see [Appendix B](#).

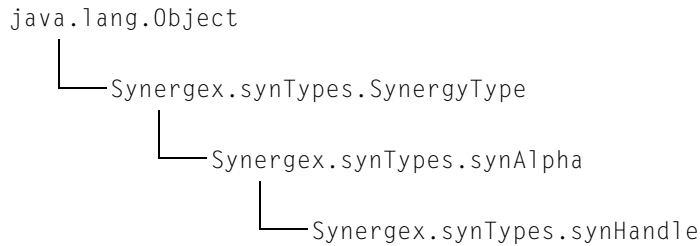
For example:

```
//declare variables
synDec    d1, d2;
long      var1 = 300;
byte      var2 = 10;

//instantiate new synDec objects to package Java variables
d1 = new synDec(var1);
d2 = new synDec(var2);
```

For more details, refer to the individual SynergyType classes in this chapter.

Synergex.synTypes.synHandle



public class synHandle extends synAlpha

A wrapper for a Java String; corresponds to a memory handle in Synergy.

Constructors public synHandle(String in)

Constructs a synHandle object representing the string.

in—the string to be stored

public synHandle()

Constructs a synHandle object with a blank string. Use this when you want to return results, but do not want to send data.

Methods public String type()

Returns a string (“HA”) that describes the SynergyType object.

Overrides type() in class synAlpha.

Usage Use the synHandle class to pass a variable length Java String that may exceed 65,535 bytes in size to Synergy. See [“Handling Variable-Length and Large Data” on page 1-12](#) for additional information and for details on how the parameter is treated by the Synergy server routine.

The following program shows how to use synHandle.

```
import java.io.*;
import java.net.InetAddress;
import Synergex.synProxy.*;
import Synergex.synTypes.*;
public class atest
{
    public static void main(String argv[])
    {
```

```
//instantiate new String to hold data
String aJavaString = new String("Really big string");

System.out.println("before call aJavaString = " +
    aJavaString);

InetAddress inetAd = null;
SynergyWebProxy swp = null;
try {
    //instantiate new synHandle
    synHandle aHdl = new synHandle(aJavaString);

    //instantiate new SynergyWebProxy
    swp = new SynergyWebProxy("elmo", 2356,
        inetAd.getLocalHost());

    //make xcall
    swp.xcall("inoutHandle",aHdl);

    //extract the returned data
    aJavaString = aHdl.value();
    swp.sendShutdownMessage(); //close connection
    swp = null;
    System.out.println("after call aJavaString = "
        + aJavaString);
}
catch (Exception e)
{
    e.printStackTrace();
}
}
```

Synergex.synTypes.synImpDec

java.lang.Object

└─ Synergex.synTypes.SynergyType

└─ Synergex.synTypes.synImpDec

public class synImpDec extends SynergyType

A wrapper for Java double and float data types; corresponds to Synergy's implied-decimal data type.

Constructors public synImpDec(double *in*)

Constructs a synImpDec object from a double.

in—the number to be stored

public synImpDec(float *in*)

Constructs a synImpDec object from a float.

in—the number to be stored

public synImpDec()

Constructs a synImpDec object containing no data. Use this when you want to return results, but do not want to send data.

Methods public double value()

Returns the number stored in this object.

public void value(double *in*)

Sets the number stored in this object. Use when passing a double.

in—the number to be stored

public void value(float *in*)

Sets the number stored in this object. Use when passing a float.

in—the number to be stored

public String type()

Returns a string (“ID”) that describes the SynergyType object.

Overrides type() in class SynergyType.

Usage Use the `synImpDec` class to translate Java floats and doubles into Synergy implied-decimal data types. The value stored in a `synImpDec` object is limited to 15 digits of precision.

For example:

```
//pass in two variables
public double myCalculation(float var1, double var2)
{
    //declare variables
    synImpDec    id1,id2;
    double       returnValue;

    //instantiate new synImpDec objects to package Java
    //variables
    id1 = new synImpDec(var1);
    id2 = new synImpDec(var2);

    //make the xcall and pass the parameters
    swp.xcall("methid0019", id1, id2);

    //extract the returned data
    var1 = (float)id1.value(); //cast return value to float
    var2 = id2.value();
    .
    .
    .
    return returnValue;
}
```

To retrieve a double from a `synImpDec` object, invoke the `value()` method. For example:

```
var1 = id1.value();
```

To retrieve a float from a `synImpDec` object, you must cast the variable to the desired Java data type. For example:

```
var1 = (float)id1.value();
```

Synergex.synTypes.synNull

java.lang.Object

└─ Synergex.synTypes.SynergyType

└─ Synergex.synTypes.synNull

public class synNull extends SynergyType

Indicates that a parameter is not being passed.

Constructors public synNull()

Constructs a synNull object.

Methods public String type()

Returns a string ("NU") that describes the SynergyType object.

Overrides type() in class SynergyType.

public String value()

Returns a string containing no characters.

Usage The synNull class acts like a placeholder and is used when you do not want to pass a parameter. Use a synNull object only when a method has more than one parameter, *and* you are *not* passing one of the parameters, *and* the parameter that you are *not* passing is followed by one that you *are* passing.

You can either declare and instantiate a synNull object and then include it with the other parameters, or just instantiate a new synNull object when you invoke the xcall() method.

For example:

```
//declare and instantiate  
synNull null01 = new synNull();
```

```
//include with other parameters  
swp.xcall("methodid0012", d1, d2, null01, a3, a4);
```

or

```
//instantiate within xcall  
swp.xcall("methodid0012", d1, d2, new synNull(), a3, a4);
```


Synergex.synTypes.SynProxyBadTypeException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.synTypes.SynProxyBadTypeException

public class SynProxyBadTypeException extends SynProxyException

Signals that a SynergyType object was created with incorrect data [e.g., synDec("abc")]. Refer to the information on synAlpha, synDec, SynImpDec, and synNull in this chapter for correct construction parameters.

For more information on errors, see [“Handling Exceptions”](#) on page 11-1.

Synergex.synTypes.SynProxyInvalidStructException

java.lang.Exception

└─ Synergex.synProxy.SynProxyException

└─ Synergex.synTypes.SynProxyInvalidStructException

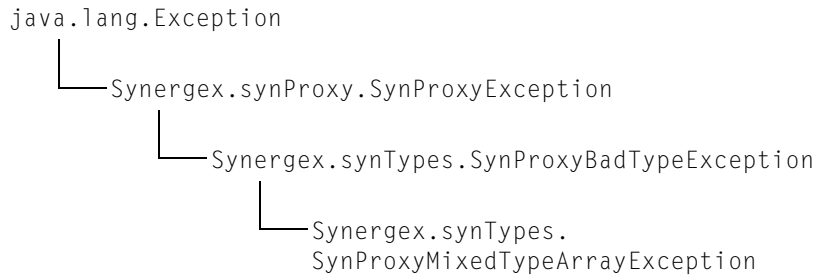
public class InvalidStructException extends SynProxyException

Signals an attempt to define or create a structure that is internally inconsistent. This exception can be thrown when a structure size does not match the sum of the defined field sizes.

You can use the `getMessage()` method to obtain the error message.

For more information on errors, see [“Handling Exceptions” on page 11-1](#).

Synergex.synTypes.SynProxyMixedTypeArrayException



public class SynProxyMixedTypeArrayException extends
SynProxyBadTypeException

Signals that a `synArray` object contains more than one type of SynergyType object (e.g., both a `synDec` object and a `synAlpha` object). A `synArray` object may contain only *one* type of SynergyType object.

For more information on errors, see [“Handling Exceptions” on page 11-1](#).

Synergex.synTypes.synStruct

java.lang.Object

└─ Synergex.synTypes.SynergyType

└─ Synergex.synTypes.synStruct

```
public class synStruct extends SynergyType
```

Represents a Synergy record or a group within a record.

The `synStruct` class is used by the Java code that is created when you generate Java class wrappers. You do not need to call this class directly.

Synergex.util.SWPConnect

java.lang.Object

└─ Synergex.util.SWPConnect

public class SWPConnect extends Object

The SWPConnect class is used by the Java code that is created when you generate Java class wrappers. This class handles the connection to and disconnection from *xfServerPlus*. It instantiates a *SynergyWebProxy* (see [page 12-8](#)), which actually makes the connection to *xfServerPlus*.

You do not need to call this class directly.

Constructors public SWPConnect(File *propFile*) throws UnknownHostException, IOException, xfServerPlusUnavailableException, SynProxyNetException, xfServerNackException
Connects to the host name on the port defined in the specified *xfNetLink* Java properties file.

propFile—the File object that names the file that contains the settings for *xf_RemoteHostName* and *xf_RemotePort*

public SWPConnect(String *host*, int *port*, int *minport*, int *maxport*, boolean *debugtrace*, String *debugtracefile*) throws UnknownHostException, IOException, xfServerPlusUnavailableException, SynProxyNetException, xfServerNackException



The port number range feature will be deprecated in a future version, which means this constructor will be deprecated also. We recommend that you use a constructor that does not pass *minport* and *maxport*.

Connects to the specified host name on the specified port. This constructor also passes the minimum and maximum port numbers for the port range and enables you to turn on logging and specify a log file.

host—the host name of the machine where *xfServerPlus* is running

port—the port number that *xfServerPlus* is listening on for remote session requests

xfNetLink Java Class Reference

SWPConnect

minport—the beginning value for the port number range that the *SWPConnect* object uses to communicate with *xfServerPlus*

maxport—the ending value for the port number range that the *SWPConnect* object uses to communicate with *xfServerPlus*

debugtrace—boolean specifying whether to display debug output (packets sent and received)

debugtracefile—filename to which the debug output will be written. You can include the full path name. If you want the output sent to the screen instead, set this to 'null'.

```
public SWPConnect(String propFile, String poolID, SWPManager  
    poolManager) throws UnknownHostException, IOException,  
    xfServerPlusUnavailableException, SynProxyNetException,  
    xfServerNackException
```

Connects to the host name and port specified in the named *xfNetLink* Java properties file. This constructor is used when connection pooling is enabled. It gets a connection from the specified pool.

propFile—a string that contains the full path of the *xfNetLink* Java properties file

poolID—the ID of the pool from which to get a connection

poolManager—the instance of the pool manager to use

Methods

```
public int getConnectCount()
```

Gets the number of connections (*SynergyWebProxy* objects).

```
public void incConnectCount()
```

Increments the connection count.

```
public SynergyWebProxy getM_swp()
```

Gets the *SynergyWebProxy*. Used when sharing a connection.

```
public void disconnect() throws Exception
```

Calls *SynergyWebProxy.sendShutdownMessage()* to close the connection.

Synergex.util.SWPManager

java.lang.Object

└─ Synergex.util.SWPManager

public class SWPManager extends Object

The SWPManager class is used to create, shut down, and otherwise manage pools of connections to *xfServerPlus*. This class uses a singleton pattern: only one instance of the class will be created in your application. For details on using Java connection pooling, see [“Understanding Java Connection Pooling” on page 10-20](#) and [“Implementing Pooling” on page 10-22](#).

Methods static synchronized public SWPManager getInstance()

Instantiates a pool of connections to *xfServerPlus* or gets an instance of an already instantiated pool. The first time this method is called, a connection pool is created using the settings in the pooling properties file. If there are multiple pools defined in the pooling properties file, all are created on the first call to `getInstance()`. Subsequent calls to `getInstance()` get an instance of the pool.

public void resetPoolProperties(String *poolID*) throws
xfPoolException

Rereads settings in the pooling properties file that are associated with the specified pool ID. Settings in the *xfNetLink* Java properties file used by that pool will also be reread. Use this method when you need to update the pooling properties file or the *xfNetLink* Java properties file and don't want to completely shut down and restart the application.

This method closes down all of the available connections in the specified pool, and then restarts the pool with the minimum number of connections and using the new settings. Connections that are in use when you run `resetPoolProperties()` will be discarded when they are released (even if `poolReturn` is set to “true”), so that all connections will use the new settings. *poolID*—the ID of the pool for which properties need to be reset

```
public void returnToMinimum(String poolID)
```

Returns the size of the pool to the minimum specified in the pooling properties file. Use this method only when `poolReturn` is set to “true” in the pooling properties file; it has no effect when `poolReturn` is set to “false”.

`poolID`—the ID of the pool that should be reset to the minimum number of connections

```
public synchronized void shutdown()
```

Closes all available connections in all the pools defined in the pooling properties file, and then destroys the pools. Connections that are in use when `shutdown()` is called will be terminated when they are released. The next call to `getInstance()` will restart the pools.

```
public SWPConnect getConnection(String poolID)
```

Returns a connection from the specified pool. If no connections are available, a null is returned. This method is called by the Java code that is generated when you create Java class wrappers. You do not need to call this method directly.

`poolID`—the ID of the pool from which to get a connection

```
public void freeConnection(String poolID, SWPConnect connection)
```

Frees the specified connection from the specified pool. This method is called by the Java code that is generated when you create Java class wrappers. You do not need to call this method directly.

`poolID`—the ID of the pool to which the connection belongs

`connection`—the instance of the connection returned with `getConnection()`

Usage See “[Using Your JAR File with Connection Pooling](#)” on page 10-22 for sample code that uses some of the methods in the `SWPManager` class.

Synergex.util.xfJCWException

java.lang.Exception

└─ Synergex.util.xfJCWException

```
public class xfJCWException extends java.lang.Exception
```

Signals that there was an exception at the Java class wrapper level. You should catch this exception if your code uses Java class wrappers.

This class catches exceptions that occur within the generated Java class wrapper and all exceptions that occur within the **xfnljav.jar** file. It serves as a wrapper for the other xfNetLink Java exceptions (except for `xfPoolException`).

Read the error message to determine what the specific problem is.

For more information on errors, see [“Handling Exceptions” on page 11-1](#).

Synergex.util.xfPoolException

java.lang.Exception

└─ Synergex.util.xfPoolException

public class xfPoolException extends java.lang.Exception

Signals that there was an exception at the Java connection pool level. You should catch this exception if your code uses Java connection pooling.

This exception can indicate any of the following situations:

- ▶ Missing or incorrect data in the pooling properties file or the *xfNetLink* Java properties file.
- ▶ Problems creating the connections in the pool, including problems with *xfServerPlus* licensing.
- ▶ Errors that occur when calling the pooling support methods, including errors within the Synergy method being called.

Read the error message to determine what the specific problem is.

For more information on pooling, see [“Understanding Java Connection Pooling” on page 10-20](#) and [“Implementing Pooling” on page 10-22](#). For more information on errors, see [“Handling Exceptions” on page 11-1](#).

Part IV: *xf*NetLink COM Edition

Creating Synergy COM Components

Using *x/NetLink* COM Edition's component generation tools, you can make type libraries that reference Synergy methods you have defined in the Synergy Method Catalog (SMC). This enables a COM-compliant client to call remote Synergy business logic.

x/NetLink COM is supported on Windows only. However, if your Synergy code is on UNIX or OpenVMS, you can modularize your code, create your ELBs or shared images, enter data in the SMC, and create a Synergy XML file before moving to Windows to create the type library.

This chapter includes a brief system overview, and then tells you how to

- create a Synergy/DE COM Component project in Workbench.
- generate and register a Synergy type library (TLB) from Workbench or from the command line.

See [chapter 14](#) for information on deploying the TLB and other files you need and using the TLB in an ASP or VB application. See [chapter 15](#) for information on errors and troubleshooting.

System Overview

[Figure 13-1 on page 13-2](#) shows the primary components of a distributed application that accesses Synergy code from a COM client. The diagram describes two machines:

- A client machine running *x/NetLink* COM and an application that uses a Synergy type library. If you're developing a two-tier system with Visual Basic, the client is the end-user's machine. If you're developing a three-tier system with Active Server Pages, the client is the web server machine.
- A Synergy server running *x/ServerPlus*, which handles the remote execution of Synergy routines. The routines are made available for remote execution by including them in an ELB or shared image and defining them in the Synergy Method Catalog (SMC), also located on the server machine. You can populate

Creating Synergy COM Components

System Overview

the SMC with routine information by entering it manually through the Method Definition Utility or by attributing your code, running **dbl2xml** to create an XML file, and then loading that file into the SMC. You may use multiple servers; each machine requires an *xfServerPlus* license.

xfNetLink COM enables you to use your existing Synergy code without rewriting it, provided that the code is already written in the form of an external subroutine or function. If any of the routines require input from or send messages to the user, or if they might generate untrapped errors, they must be adjusted to work as server-level logic.

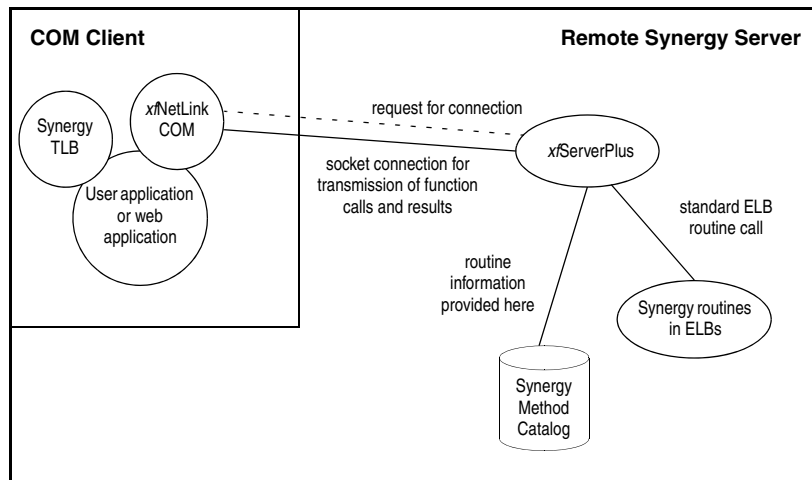


Figure 13-1. Accessing Synergy from COM.

The Big Picture

This section lists all the steps that need to be completed to create a distributed system using COM. Note that these steps may not all be done by you, the Synergy developer. For example, you may create the type library and then give it to an ASP developer to create a web front-end for your application. The ASP developer may work with a web page developer to create the HTML portions of the user interface. However, this section should give you a feel for everything that needs to be done, regardless of who does it.

1. Install the necessary Synergy software. The components you need to install vary depending on your set-up (e.g., what OS your source files are on, where you intend to do the development, and so on). Note the following:
 - ▶ The type library generation tools are part of the *xfNetLink* COM installation and must be installed on a Windows system.
 - ▶ The Professional Series Development Environment (or Workbench) installation includes the SMC/ELB comparison utility, the MDU and SMC files, **genxml**, and the test skeleton generation utility. On UNIX and OpenVMS, these utilities are part of the Synergy Language installation.
 - ▶ *xfServerPlus* must be installed on your Synergy server machine. The *xfServerPlus* (*xfSeries*) installation also includes the MDU, the SMC files, and **genxml**. For detailed steps on setting up *xfServerPlus*, see “[The Big Picture](#)” on page 3-1.
2. Modularize your existing Synergy code for the routines that you want to access remotely and encapsulate them in an ELB. See [chapter 1](#), “[Preparing Your Synergy Server Code](#).”
3. Populate the Synergy Method Catalog (SMC) with information about your Synergy routines. You can do this by attributing your code or by entering data with the Method Definition Utility. As you do this, you’ll group the routines into interfaces. If you choose to put the SMC somewhere other than DBLDIR, set the XFPL_SMC_PATH environment variable. See [chapter 2](#), “[Defining Your Synergy Methods](#).”
4. In the **xfpl.ini** file, set logging options for the *xfServerPlus* log and set logicals that point to the ELBs you specified in the SMC (if necessary). You may also need to set other options in the **xfpl.ini** file; see [Appendix A](#) for a complete list of **xfpl.ini** configuration settings.

Creating Synergy COM Components

The Big Picture

If you choose to put the **xfpl.ini** file somewhere other than DBLDIR, set the XFPL_INIPATH environment variable.

See [chapter 3, “Configuring and Running xfServerPlus,”](#) for information on the log and XFPL_INIPATH. See [“Defining Logicals” on page 1-4](#) for information on setting logicals that point to your ELBs.

5. Create a user account on the *xfServerPlus* machine to run *xfServerPlus* sessions and then start *xfServerPlus*. See [“Running xfServerPlus” on page 3-3](#).
6. If you’re developing on Windows, create a Synergy/DE COM Component project in Workbench. Specify the component information: the name and location of the type library, the location of the SMC, the location of repository files (if applicable), the interfaces you want to include, and (if desired) alternate names for those interfaces. See [“Creating a Synergy/DE COM Component Project” on page 13-5](#) for details.
7. If you’re developing on Windows, generate and register the type library by building the COM component project in Workbench. See [“Generating a Type Library” on page 13-8](#).

If your Synergy code is on UNIX or OpenVMS, run the **genxml** utility to create a Synergy XML file. Then, move the XML file to your Windows system and use **gentlb** to create the type library and **regtlb** to register it. See [“Creating and Registering a Type Library from the Command Line” on page 13-11](#).

8. Distribute the TLB and other necessary files to the COM developer, who will then use the component when writing the UI code for your distributed application. See [“Deploying the Client” on page 14-1](#) and [“Using Your Type Library” on page 14-6](#).
9. Test your distributed system.

Creating and Registering a Type Library in Workbench

The component generation tools enable you to create a COM type library of Synergy methods. This type library can be used with VB, ASP, and other COM-compliant client applications to invoke methods that are described in the type library. To create and register the type library, you need to

- ▶ create a Synergy/DE COM Component project in Workbench and specify information about how the type library should be constructed.
- ▶ generate the type library by building the project.
- ▶ register the type library.



You can generate and register the type library from the command line rather than from Workbench. If your Synergy code is on UNIX or OpenVMS, you can generate the XML file there, and then move the file to Windows to create the type library. See [“Creating and Registering a Type Library from the Command Line” on page 13-11](#).

Creating a Synergy/DE COM Component Project

1. In Workbench, select Project > New, and then select Synergy/DE COM Component from the list of project types. (Expand the Synergy/DE node in the Project type display to select Synergy/DE COM Component.)
2. Give the project a name in the Project name field, and indicate whether you want to create a new workspace or add the project to an existing workspace. For more information on using this dialog, as well as information about basic Workbench features, see the Workbench online help or the [“Developing Your Application in Workbench”](#) chapter of *Getting Started with Synergy/DE*.
3. Specify the following component information for the new project in the Component Information dialog box. See [figure 13-2](#). This dialog displays automatically when you create a new COM component project. If you need to display it later, select Build > Component Information.

Name. Enter a name for the type library (the .tlb file). This is the name that users will see when they register your type library. The default is the project name.

Directory. Specify the directory to which you want the TLB saved. The default is the project location. If you enter a logical in this field, it must be followed by a colon (e.g., DBLDIR:).

Creating Synergy COM Components

Creating and Registering a Type Library in Workbench

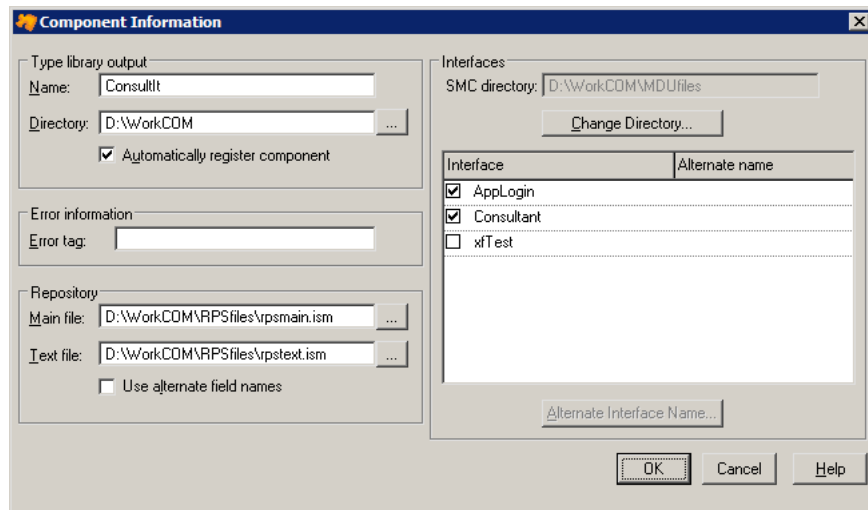


Figure 13-2. The Component Information dialog box for a Synergy/DE COM Component project.

Automatically register component. Indicate whether you want the type library to be automatically registered on the machine that you are building it on. You can also register it manually, or register it on a different machine; see [“Registering a Type Library” on page 13-8](#). Type libraries must be registered in the Windows registry before they can be accessed by other programs.

Error tag. (optional) Specify the text that you want appended to the beginning of errors generated by this TLB. The error tag is optional; it can help you identify which component threw the error. Errors generated by Synergy routines called from COM are formatted thus

```
<error tag>:<TLB name>:<interface name>:<error message>
```

Repository main file. (optional) If any of the methods that will be included in this TLB pass structures as parameters, specify the location of the repository main file for those structures. This must match the repository that was used when entering data in the SMC. If you specify a main file, you must also specify a text file.

The default is the value of the environment variable RPSMFIL. If it is not defined, the default is **RPSDAT:rpsmain.ism**. If neither RPSMFIL nor RPSDAT are defined, the default is **rpsmain.ism** in the path specified in the Working directory property of the project. If the Working directory is not defined, the default is **rpsmain.ism** in the location where the project is stored.

Repository text file. (optional) If any of the methods that will be included in this TLB pass structures as parameters, specify the location of the repository text file for those structures. This must match the repository that was used when entering data in the SMC. If you specify a text file, you must also specify a main file.

The default is the value of the environment variable RPSTFIL. If it is not defined, the default is **RPSDAT:rpstext.ism**. If neither RPSTFIL nor RPSDAT are defined, the default is **rpstext.ism** in the path specified in the Working directory property of the project. If the Working directory is not defined, the default is **rpstext.ism** in the location where the project is stored.

Use alternate field names. Indicate whether you want to use the value in the Alternate name field in Repository instead of the value in the Name field as the property name. If not selected, the field name in the structure becomes the property name in the type library. If selected, the value in the Alternate name field is used if it exists; else, the value in the Name field is used.

SMC directory. This field displays the path for the Synergy Method Catalog that this type library uses. If set, Workbench will use XFPL_SMCPATH to find the SMC files; else, the default is DBLDIR. To change the SMC directory, click the Change Directory button to display the Browse for SMC Directory dialog box. Navigate to the directory, double-click to select it, and click OK. The selected path will display in the SMC directory field and the list of interfaces will be refreshed, displaying all interfaces in the selected SMC.

Interfaces. Select the interfaces you want to include in the type library by clicking in the box to the left of the interface name.

Alternate name. (optional) You can provide an alternate name for any interface you select. To specify an alternate name, highlight a selected interface, and then click the Alternate Interface Name button and enter another name.

You may wish to use an alternate name if the interface name in the SMC is not what you want users to see as the interface name in the TLB. In addition, if your interface names differ only in case, or if you have a structure with the same name as an interface, you can use the alternate interface name to avoid having numbers appended to the names. See the note on [page 2-25](#).

To remove an alternate interface name, clear the checkbox for the interface and then reselect it.

4. Click OK in the Component Information dialog box.

Generating a Type Library

To generate the type library, select Build > Build Type Library.

This will run the SMC/ELB Comparison utility (`smc_elb.exe`; see [page 2-55](#) for more information), create the type library (`.tlb` file), and—if you checked “Automatically register component” in the Component Information dialog box—register it as well.

If the type library already exists, the TLB generation utility will update it. If the type library is in use, you’ll see an error. It may be the case that the web server, Visual Basic, or the pool (if you’re using COM pooling) has the type library locked. Try shutting down the web server or VB and building the type library again. If you are using COM pooling, you must stop the pool before rebuilding the type library. See [“Changing the Pool Configuration” on page 14-21](#) for information on stopping and restarting the pool.

Registering a Type Library

The type library must be registered in the Windows registry on any machine on which you want to use it. If you selected “Automatically register component” in the Component Information dialog box, the type library was registered on the machine on which you built it.

If the type library was not registered automatically, follow these instructions to register it from Workbench.

1. Open the Synergy/DE COM Component project that is associated with the type library you want to register.
2. Press ESCAPE to access the Workbench command line and then enter the command following command:

```
SynRegTLB
```

Unregistering a Type Library

Follow these instructions to unregister a type library from Workbench.

1. Open the Synergy/DE COM Component project that is associated with the type library you want to unregister.
2. Press ESCAPE to access the Workbench command line, and then enter the following command:

```
SynRegTLB -u
```



You can also register a type library from the Synergy Type Library Configuration utility, and you can both register and unregister a type library from the (DOS) command line. You'll need to use one of these methods when you register TLBs on machines that do not have Workbench installed. See ["Using the Synergy Type Library Configuration Utility" on page 14-2](#) and ["The regtlb Utility" on page 13-16](#).

Viewing a Type Library in Workbench

After you generate the type library, it will be added to your COM component project. On the Projects tab in the project toolbar, expand the Type Library Files node to see it.

To view the contents of the TLB in Workbench, go to the Symbols tab in the project toolbar and expand the Packages/Namespace node under the Workspace node. You'll see a node for each TLB in the project. Expand the TLB node to see nodes for each procedural interface and structure interface in the TLB. If you specified an alternate interface name in the Component Information dialog box, you'll see that name in the Symbols tab.

Expand the node of a structure interface to see the fields (properties) in the structure. Expand the node of a procedural interface to see the methods in that interface—both your own methods and the nine *x/NetLink* COM utility methods. See [figure 13-3 on page 13-10](#).

Creating Synergy COM Components

Creating and Registering a Type Library in Workbench

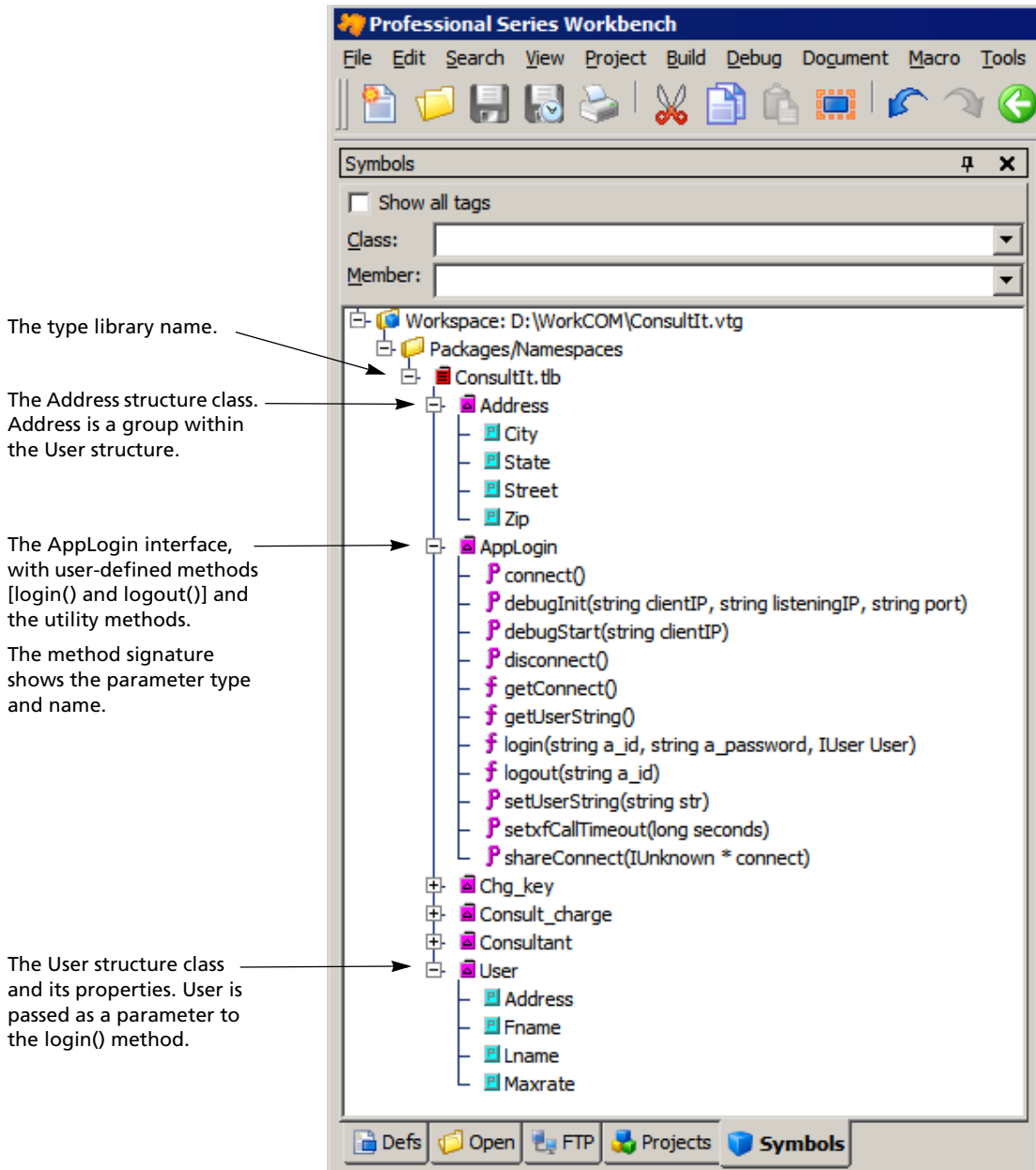


Figure 13-3. Viewing a type library in Workbench.

Creating and Registering a Type Library from the Command Line

You can create and register a type library from the command line using the utilities described in this section. You'll need to do the following:

1. Use the **genxml** utility to create a Synergy XML file. See [“The genxml Utility,”](#) below.
2. If you generated the XML file on a UNIX or OpenVMS system, move it to a Windows system.
3. Use the **gentlb** utility to create the type library. See [“The gentlb Utility”](#) on [page 13-15](#).
4. Use the **regtlb** utility to register the type library. See [“The regtlb Utility”](#) on [page 13-16](#).



If you are updating an existing type library from the command line, unregister it before regenerating and reregistering. This will help keep your registry clean.

The genxml Utility

The **genxml** utility creates a Synergy XML file from SMC method definitions and repository structure definitions. This is an intermediate step in creating a type library.

This utility runs on all supported Synergy/DE platforms, so you can create a type library regardless of whether you develop your Synergy applications on Windows, UNIX, or OpenVMS. **Genxml** is installed in the DBLDIR directory. If you are on UNIX or OpenVMS, after generating the Synergy XML file, you will need to move it to a Windows system to run **gentlb** and create the type library.



The **genxml** utility checks structure sizes in the SMC against the corresponding structures in the repository and reports a warning if there are discrepancies. Although the XML file is generated anyway, you should use the MDU's Verify Catalog utility to update the structure sizes in the SMC. (See [“Verifying Repository Structure Sizes and Enumerations”](#) on [page 2-42](#).) Failure to do so can cause errors at runtime (because the structure information in the component will differ from that in the SMC).

Creating Synergy COM Components

Creating and Registering a Type Library from the Command Line

Syntax `dbr genxml -f xmlFilename -i intName [-a altIntName]
[-d targetDir] [-s smcDir] [-m rpsMain -t rpsText] [-n]
[-v msgLevel] [-e errTag] [-?]`

Arguments `-f xmlFilename`

The name to use for the XML file (and, subsequently, the type library). You can include the complete path if desired. The extension “.xml” will be appended to this filename if you don’t specify an extension.

`-i intName`

Name of the interface from the SMC to include in the XML file. You may pass multiple interface names; each must be preceded with the `-i` option. Remember, the interface name is case sensitive.

`-a altIntName`

(optional) Alternate interface name. Use this name for the interface previously specified with the `-i` option. **Genxml** uses the associated `-i` interface to pull methods from the SMC; the alternate name is included in the XML file and is used as the class name when **gentlb** is run. If you pass multiple interface names, each may have an alternate name. Each alternate name must be preceded with the `-a` option. See the example on [page 13-14](#).



When using alternate names, sequence matters. The `-a` option must follow the `-i` option that it applies to. You may specify multiple interface names, and each may have an associated alternate name immediately following it.

`-d targetDir`

(optional) The target directory for the XML file. If not passed, the XML file is created in the directory specified in the `-f` option. If no directory is specified with `-f`, the file is created in the current directory.

`-s smcDir`

(optional) Directory where the SMC files (`cdt.is?` and `cmpdt.is?`) are located. If not passed, DBLDIR is used.

`-m rpsMain`

(optional) Full path to the repository main file that contains the structures referenced in the SMC. Use with `-t`. This option is used only if you are passing structures as parameters. If not passed, **genxml** uses the environment variable

Creating Synergy COM Components

Creating and Registering a Type Library from the Command Line

RPSMFIL to determine the name of the repository main file; if that is not set, it uses **RPSDAT:rpsmain.ism**. If RPSDAT isn't set, **genxml** looks in the current directory for **rpsmain.ism**.

-t *rpsText*

(optional) Full path to the repository text file that contains the structures referenced in the SMC. Use with **-m**. This option is used only if you are passing structures as parameters. If not passed, **genxml** uses the environment variable RPSTFIL to determine the name of the repository text file; if that is not set, it uses **RPSDAT:rpstext.ism**. If RPSDAT isn't set, **genxml** looks in the current directory for **rpstext.ism**.

-n

(optional) Indicates that you want to use the value in the Repository Alternate name field instead of the value in the Name field. This option pertains only if you are passing structures as parameters. If not passed, the field name in the structure becomes the property name in the type library. If passed, the value in the Alternate name field is used if it exists; else, the value in the Name field is used.

-v *msgLevel*

(optional) Level of verbosity in messages.

0 = no messages

1 = error messages and warnings

2 = everything included in level 1, plus success messages (default)

3 = everything included in level 2, plus return codes and the location of the SMC and repository files

-e *errTag*

(optional) A text string that will be appended to the beginning of all errors generated by the type library. Put the string in quotation marks. The error tag can help you identify which component threw the error. See [page 13-6](#) for the error message format.

-?

(optional) Displays a list of options and the version number for **genxml**.

Creating Synergy COM Components

Creating and Registering a Type Library from the Command Line

Examples This example creates a Synergy XML file named **myLib.xml**, which includes two interfaces (OR and RT). Both interfaces use alternate names (Orders and Returns, respectively). The target directory for the XML file is c:\temp, which is also where the SMC files are located. The error tag text is “myLibError”.

For example, on Windows:

```
dbr DBLDIR:genxml -f myLib -i OR -a Orders -i RT -a Returns  
-d c:\temp -s c:\temp -e "myLibError"
```

On OpenVMS, you'll need to define **genxml** as a foreign command and then execute it. In the example below, we have quoted the mixed case strings to preserve the case.

```
$ GENXML:==$DBLDIR:GENXML  
$ GENXML -F "myLib" -I OR -A "Orders" -I RT -A "Returns"-  
-D SYS$SCRATCH: -S SYS$SCRATCH: -E "myLibError"
```

The gentlb Utility

The **gentlb** utility uses the Synergy XML file created by **genxml** and creates a type library (.tlb file). **Gentlb** runs on Windows only; it is installed in the xfnlcom directory.



If you are regenerating an existing type library, make sure it's not in use. If it is in use, you'll get an error message stating that the file cannot be saved. It may be the case that the web server, Visual Basic, or the pool (if you're using pooling) has the type library locked. Try shutting down the web server or VB and running **gentlb** again. If you are using pooling, you must stop the pool before rerunning **gentlb**. See ["Changing the Pool Configuration" on page 14-21](#) for information on stopping and restarting the pool.

Syntax `gentlb -f xmlFilename [-d targetDir] [-v msgLevel] [-?]`

Arguments `-f xmlFilename`

The name of the XML file generated with **genxml**. Include the full path if necessary; if the path is not included, the current directory is assumed.

`-d targetDir`

(optional) Specifies the directory where the .tlb file will be created. If not passed, and a directory was specified with `-f`, the .tlb file will be created in the same directory as the XML file. If no directory was specified with `-f`, it is created in the current directory.

`-v msgLevel`

(optional) Level of verbosity in messages.

0 = no messages

1 = error messages

2 = error and success messages (default)

3 = everything included in level 2, plus return codes and the location of the SMC and repository files

`-?`

(optional) Displays a list of options and the version number for **gentlb**.

Examples This example creates a type library from **myLib.xml**, located in the c:\temp directory, and puts it in the directory c:\myDir. The new file will be named **myLib.tlb**.

```
gentlb -f c:\temp\myLib.xml -d c:\myDir
```

The regtlb Utility

The **regtlb** utility registers or unregisters a type library. **Regtlb** runs on Windows only; it is installed in the xfnLCOM directory.

The type library needs to be registered on every machine on which it is used, such as your web server (for ASP) or the end-user's machine (for VB). You can also use the Synergy Type Library Configuration Utility to register a type library. See [“Deploying the Client” on page 14-1](#) for more information.

Syntax `regtlb -f tlbFilename [-d targetDir] [-u] [-v msgLevel] [-?]`

Arguments `-f tlbFilename`

The name of the type library to register. Use the full path unless the TLB is in the current directory or the path is specified with **-d**.

`-d targetDir`

(optional) The directory where the TLB is located. If not passed, and a directory was specified with **-f**, **regtlb** looks for the TLB in that directory. If no directory was specified with **-f**, **regtlb** looks for the TLB in the current directory.

`-u`

(optional) Unregisters the specified type library. If **-u** is not specified, the type library is registered.

`-v msgLevel`

(optional) Level of verbosity in messages.

0 = no messages

1 = error messages

2 = error and success messages (default)

3 = everything included in level 2, plus return codes and the location of the SMC and repository files

`-?`

(optional) Displays a list of options and the version number for **regtlb**.

Examples The first example below registers the type library **myLib.tlb** located in the temp directory. The second example unregisters the same type library.

```
regtlb -f myLib.tlb -d c:\temp
```

```
regtlb -f myLib.tlb -d c:\temp -u
```

Calling Synergy Routines from COM

This chapter explains how to deploy the type library and other tools necessary to create a distributed application with *xfNetLink* COM Edition and Synergy, and how to use the type library. It also includes instructions on using COM pooling, as well as a method reference for the utility methods included in your type library.

Deploying Your Distributed Application

This section tells you what needs to be installed and configured on the *xfServerPlus* machine and the client machine. You will need to refer to this section when deploying your type libraries so that your ASP or VB developer can use them, as well as when creating the installation program for your distributed application.

Deploying the Server

For instructions on deploying the server portion of your distributed application, see [“Deploying the Server” on page 3-44](#).

Deploying the Client

1. Install *xfNetLink* COM.
2. Copy your TLB(s) to the client. For the sake of convenience, you may want to put them in the *xfNLCOM* directory or create a subdirectory for them within *xfNLCOM*.
3. Use the Synergy Type Library Configuration utility (**syntlbcfg.exe**) to register and configure your Synergy TLBs. See [“Using the Synergy Type Library Configuration Utility” on page 14-2](#) below. You can also register Synergy type libraries from the command line; see [“The regtlb Utility” on page 13-16](#).
4. If you are using COM pooling, create, configure, and start the pool. See [“Setting Up the Client Machine” on page 14-17](#) for instructions.

Using the Synergy Type Library Configuration Utility

The Synergy Type Library Configuration utility (**syntlbcfg.exe**) can be used to register and configure type libraries on the client system. It is installed in the **xfNLCOM** directory.

You can start the Synergy TLB Configuration utility from the command line, the Windows Start menu (Programs > Synergy/DE > Utilities), or, if Workbench is installed, from Synergy/DE > Utilities.



Because the Synergy TLB Configuration utility writes to the Windows registry, you must be logged on as a member of the administrator group to use the utility.

► To register a Synergy TLB

This procedure registers a Synergy type library in the Windows registry. The TLB must be registered on every machine on which it is used.

1. Click the Add TLB button.
2. Type the TLB name, or click the browse button and select it.
3. Click OK. You'll see a message if it is registered successfully, and the TLB will display in the list of Synergy interfaces in the format *TLBName.interfaceName*.
4. Repeat this procedure for each Synergy TLB.



You cannot unregister a type library from the Synergy TLB Configuration utility. To unregister a TLB, run **regtlb** from the command line with the **-u** option (see [“The regtlb Utility” on page 13-16](#)) or use Workbench (if installed; see [“Unregistering a Type Library” on page 13-9](#)).

► To configure a Synergy interface

The values set for the “Default” interface apply to all interfaces in all of your Synergy type libraries. You must specify at least **xfhost** and **xfport** for the Default interface. You can also specify some or all of the interface settings at the component interface level; settings at this level override the default settings. Values set in this utility are written to the Windows registry.

1. Select the interface from the list of Synergy Interfaces.
2. Click the Add button to display the Add Interface Setting dialog box.

3. In the Key field, select the setting you want to add.
4. In the Value field, specify the value for the key you selected. Refer to the table on [page 14-4](#).
5. Click the Save button.
6. Repeat this procedure for each setting you want to add.

► **To use the default settings**

Follow these instructions if you have configured settings for a particular interface, but now want it to use the default settings. This procedure deletes all settings for the interface.

1. Select the interface from the list of Synergy Interfaces.
2. Click the Use Defaults button.
3. Click Yes at the confirmation message.

► **To modify a setting**

1. Select the interface from the list of Synergy Interfaces. The current settings for that interface will display below in the Interface Settings list.
2. In the Interface Settings list, select the setting you want to change.
3. Click the Modify button.
4. In the Modify Interface Setting dialog box, type a new value for the key. See the table on [page 14-4](#) for details on setting values for the various keys.
5. Click Save.

► **To delete a setting**

Deleting a setting from an interface causes the interface to use the default value for that setting. If you want to delete all settings, see [“To use the default settings” on page 14-3](#).

1. Select the interface from the list of Synergy interfaces.
2. In the Interface Settings list, select the setting you want to delete.
3. Click the Delete button, and then click Yes at the confirmation message.

Calling Synergy Routines from COM

Using the Synergy Type Library Configuration Utility

Interface Settings	
Key	Set value to
connect timeout	The number of seconds you want <i>xfNetLink</i> to wait for an acknowledgment from the session started by the logic server in <i>xfServerPlus</i> . (This is 'B' in figure 5-1 on page 5-4 .) The default for normal operation is 120 seconds; for debug it is 600 seconds. Both the normal time-out and the debug time-out will change to the value you specify here. This value is also used to set the request for session time-out ('A' in figure 5-1 on page 5-4).
initialize timeout	(pooling only) The number of seconds you want <i>xfNetLink</i> to wait for a return from a remote call to <i>xfServerPlus</i> when the pool is being started. This value controls a special call time-out that applies only to the <i>Initialize()</i> method when you are using COM object pooling. The default is 30 seconds. The call time-out is reset to 30 minutes after the <i>Initialize()</i> method is called. See "Setting a Call Time-Out" on page 14-12 for information on the regular call time-out.
logfile	The path and filename of the file you want to use for logging.
logging	The information you want logged. When you select logging in the Key field, the logging options become available. Check the boxes to indicate what information you want logged. See page 15-10 for a sample log. ▶ Show errors. Logs all fields of an Error object when an error is returned. ▶ Show user method calls. Logs calls to methods in the SMC. Includes parameters passed in and shows what is in the parameter after the call. ▶ Show xf support method calls. Logs calls to the utility methods (e.g., <i>connect()</i>). Includes parameters passed in and out. ▶ Show property method calls. Logs the setting of field values for properties of structure interfaces. Applicable only if you are passing structures as parameters. Valid only when set for the Default interface. ▶ Show IStructManager calls. Logs information about internal structure processing. Use for debugging when requested by Synergy/DE Developer Support. Valid only when set for the Default interface. ▶ Show interface creation/query calls. Logs the creation and destruction of objects. Use for debugging when requested by Synergy/DE Developer Support. ▶ Show packet information. Logs the complete packets that are sent and received on the client.

Interface Settings (continued)	
Key	Set value to
maxport	The upper end of the range of ports that you want the client to use to communicate with <i>xfServerPlus</i>. If you specify maxport, you must also specify a value for minport. The maxport value must be greater than 1024 and greater than or equal to minport. Be sure to specify a range that is large enough. If you do not specify a range, ports in the range 1024 through 65534 are selected at random. Note: If your client application is a web application, and there is a firewall between your web server and the system running <i>xfServerPlus</i>, you must set up packet filter rules that allow packets to be exchanged by these two systems on the range of ports specified with minport and maxport.  The port number range feature will be deprecated in a future version; we do not recommend using the minport and maxport settings.
minport	The lower end of the range of ports that you want the client to use to communicate with <i>xfServerPlus</i>. If you specify minport, you must also specify a value for maxport. The minport value must be greater than 1024 and less than or equal to maxport. See also the description for maxport.
pool return	(pooling only) "true" if you want the objects instantiated from this interface returned to the pool. The default is "false". See "Reusing Connections and Objects" on page 14-15 for more information.
xfhost	(required for Default) The name or IP address of your <i>xfServerPlus</i> machine. You can set this value to "localhost" if <i>xfServerPlus</i> is on the same machine as <i>xfNetLink</i>.
xfport	(required for Default) The port that <i>xfServerPlus</i> is listening on. The port must be in the range 1024 through 65534. The default port for <i>xfServerPlus</i> is 2356.

Using Your Type Library

Once you've created and registered your TLB, it is ready for use in ASP or VB programs. This section contains the information that the person using your COM type library for development needs to know. If you are using pooling, also refer to [“Programming Tips for Object Pooling” on page 14-25](#).

The machine used by the COM developer must be properly configured. See [“Deploying the Client” on page 14-1](#).

1. Associate the component with your project

Before using the component you must associate it with your project in your development environment. The method for doing this depends on the environment. In Workbench, first make sure your project is the active project. Then, select Project > Project Properties. On the Files tab, click the Add Files button, navigate to the TLB file, and add it.

You can view the methods in the TLB using the object/class browser supplied with your development environment. In Workbench, go to the Symbols tab in the project toolbar and expand the Packages/Namespaces node under the Workspace node. (For more information, see [“Viewing a Type Library in Workbench” on page 13-9](#).)

When you expand the node of one of the procedural interfaces within the TLB, you'll see your own methods, along with the following *x/NetLink* COM utility methods:

```
connect()  
disconnect()  
getConnect()  
shareConnect()  
setUserString()  
getUserString()  
setxfCallTimeout()  
debugInit()  
debugStart()
```

The utility methods enable you to establish a connection with *x/ServerPlus* and perform other utility functions, such as setting a time-out and running a debug session.

2. Instantiate the COM interface

Your Synergy COM component can be instantiated using standard COM mechanisms for ASP and VB. The examples below show how to instantiate an object named “LoginObj”. (These examples refer to the type library and methods illustrated in [figure 13-3 on page 13-10](#).)

In ASP, you can use the <OBJECT> tag or CreateObject(). The examples below use VBScript.

- ▶ Use the <OBJECT> tag in a **global.asa** file.

```
<object runat=server scope=session id=LoginObj  
progid="ConsultIt.AppLogin"> </OBJECT>
```

- ▶ Use the <OBJECT> tag in an ASP page.

```
<object runat=server id=LoginObj progid="ConsultIt.AppLogin">  
</OBJECT>
```

- ▶ Use CreateObject(). Saving the LoginObj object in the Session object enables other ASP pages to use the LoginObj object.

```
Set LoginObj = CreateObject("ConsultIt.AppLogin")  
Set Session("SesLoginObj") = LoginObj
```

- ▶ Use Server.CreateObject(). With this method, the Server object creates the object and can keep track of it. Microsoft recommends using Server.CreateObject().

```
Set LoginObj = Server.CreateObject("ConsultIt.AppLogin")
```

In VB, you can use CreateObject() for both procedural and structure interfaces when the object is declared as an object. You can also use New, but only for procedural interfaces. Procedural interfaces are objects that have methods, such as AppLogin in our example. Structure interfaces are objects that contain only properties, which represent fields in a structure, such as User in our example (see [“Using Structures” on page 14-10](#)). See the note below for details on why you must make this distinction.

- ▶ Declare the variable as an object and use CreateObject(). You can use this method for both procedural and structure interfaces.

```
Dim LoginObj As Object  
Set LoginObj = CreateObject("ConsultIt.AppLogin")
```

Calling Synergy Routines from COM

Using Your Type Library

- Declare the variable and use `CreateObject()`. Use this method for procedural interfaces only.

```
Dim LoginObj As ConsultIt.AppLogin  
Set LoginObj = CreateObject("ConsultIt.AppLogin")
```

- Use `New` to declare and instantiate the object. Use this method for procedural interfaces only.

```
Dim LoginObj As New ConsultIt.AppLogin
```



`xfNetLink` COM supports late binding for all objects and early binding only on procedural interfaces. Late binding means that VB cannot determine at compile time what sort of object reference the variable contains. Instead, it determines at runtime whether the actual object has the properties and methods you call using the variable. The first VB example above uses late binding. Early binding means that VB determines at compile time what sort of object reference the variable contains. The second and third VB examples above use early binding. You cannot use early binding for structure interfaces because `xfNetLink` COM does not fully support the way the VB compiler handles arrays within a structure.

3. Connect to `xfServerPlus`

There are several ways to establish a connection with `xfServerPlus`:

- Use `connect()`. This is the recommended method. It enables you to make several calls using the same connection, and then disconnect. There are several advantages to this method: it offers improved performance; you can maintain state between calls; and you can share this type of connection with other components (see next bullet).

For example, in VB or VBScript:

```
Call LoginObj.connect()
```

- Share a connection. Two or more objects can share a connection. This method improves performance because several objects are sharing the same `xfServerPlus` session rather than each object making its own connection. To share a connection, you must first establish it with `connect()`, and then use the `getConnect()` and `shareConnect()` methods. In the example below, a connection is established for the `LoginObj` object, which the `ConsultObj` object then shares.

For example, in VB or VBScript:

```
Set LoginObj = CreateObject("ConsultIt.AppLogin")
Set ConsultObj = CreateObject("ConsultIt.Consultant")
Call LoginObj.connect()
Call ConsultObj.shareConnect(LoginObj.getConnect())
```

Or, if you will be sharing the connection among several objects, you can instantiate an object (xyz in the example below) to hold the connection, and then pass that object when you call shareConnect(). You can pass the same xyz object multiple times to share the connection among several objects, so that you don't have to repeatedly use getConnect(). Note: This method works only in ASP; in VB, use the method shown in the previous example.

For example, in VBScript:

```
Set LoginObj = CreateObject("ConsultIt.AppLogin")
Set ConsultObj = CreateObject("ConsultIt.Consultant")
Call LoginObj.connect()
Dim xyz As Object
Set xyz = LoginObj.getConnect()
Call ConsultObj.shareConnect(xyz)
```

- Create the connection automatically. When you make a call using one of the Synergy methods in your TLB, the connection is created automatically and then disconnected when the call is complete. This is the easiest method to use because it makes access to *x/ServerPlus* completely transparent. However, this method does not allow you to maintain state between calls and requires more overhead because a connection is opened and closed for each call. If you use this method and experience performance problems, you may want to use an explicit connect() instead.

In the example below, the LoginObj object is instantiated, and then the login() method is called, without first calling the connect() method.

```
Set LoginObj = CreateObject("ConsultIt.AppLogin")
Call LoginObj.login(userID, password)
```

Calling Synergy Routines from COM

Using Your Type Library

4. Invoke methods in the component

Make calls to your Synergy methods and pass the necessary parameters. Required input parameters must be initialized.

For example, to invoke the `login()` method in the `LoginObj` object:

```
Dim userID As String
Dim password As String
userID = "MFranklin"
password = "123abc"
Call LoginObj.login(userID, password)
```

5. Disconnect from *xfServerPlus*

If you connected to *xfServerPlus* using the `connect()` method, you must disconnect using the `disconnect()` method. If you have multiple objects sharing a connection, *xfServerPlus* will not completely close the connection and release the license until all objects are disconnected. For example:

```
Call LoginObj.disconnect()
```

6. Release the object

When the object is no longer needed, set it to nothing to free up system resources. If two or more objects are sharing a connection, each of them should be released.

For example, in VB or VBScript:

```
Set LoginObj = Nothing
```

Using Structures

Repository structures and groups included in a type library will appear as classes when you view the TLB. The fields in the structure will be properties of the class.

For example, say you have the following repository structure, which is passed as the third parameter to the `login()` method.

```
user
  fname      ,a25
  lname      ,a25
  maxrate    ,d18.2
  group      address ,a
    street    ,a20
    city      ,a20
    state     ,a2
    zip       ,d9
  endgroup
```

In the Workbench Symbols tab, you'll see a class named "User" with four properties—Fname, Lname, Maxrate, and Address. You'll also see a class named "Address" in the type library. ("Address" is the name of the field in the repository that references this group.) Address has four properties—Street, City, State, and Zip. (See [figure 13-3 on page 13-10](#).) Under normal circumstances, you won't need to access the Address class directly. When you instantiate a new User object, an Address object is automatically instantiated. (Fields defined as struct data type in the repository are treated the same as group fields.)



By default, the properties are named with the repository field names. See ["Passing Structures as Parameters" on page 1-8](#) for details on overriding the default name.

To access these properties, instantiate an object for the User class and assign values to the properties. (See [step 2 on page 14-7](#) for details on why you should use this method to instantiate an object for a class created from a structure.) For example:

```
Dim User1 As Object
Set User1 = CreateObject("ConsultIt.User")
User1.Fname = "Mickey"
User1.Lname = "Franklin"
User1.Maxrate = 150.00
```

To access the properties in the Address group, do this:

```
User1.Address.Street = "2330 Gold Meadow Way"
User1.Address.City = "Gold River"
User1.Address.State = "CA"
User1.Address.Zip = 956704471
```

Then, pass the User1 object when you call the login() method:

```
Call LoginObj.login(userID, password, User1)
```

See ["Passing Structures as Parameters" on page 1-8](#) for additional information about structures, including information on how overlays are handled.

Setting a Call Time-Out



You must explicitly create a connection with the `connect()` method to use `setxfCallTimeout()`.

Use the `setxfCallTimeout()` method to set the call time-out value in seconds. The call time-out measures the length of time that the COM client waits for a return from a remote call to `xfServerPlus` (this is 'C' in [figure 5-1 on page 5-4](#)). This time-out is measured for *each* send–receive request between the client and `xfServerPlus`. The default value is 1800 seconds (30 minutes).

Once this value has been set, it will continue to be used for all subsequent calls in the current session until it is reset with another invocation of this method.

For example, to set the call time-out to 20 minutes, use

```
Call LoginObj.setxfCallTimeout(1200)
```

To set a connect time-out value, see [“Using the Synergy Type Library Configuration Utility” on page 14-2](#). To set a call time-out value for `xfServerPlus`, see [SET_XFPL_TIMEOUT on page 1-31](#).



When you use COM pooling, there is a special call time-out that applies to the `Initialize()` method only. For information on the initialize time-out, see [“Using the Synergy Type Library Configuration Utility” on page 14-2](#).

Writing to the `xfServerPlus` Log



You must explicitly create a connection with the `connect()` method to use `setxfCallTimeout()`.

Use the `setUserString()` method to pass a user string that is written in the `xfServerPlus` log. This string is stored, and the value set can be retrieved with `getUserString()`. You can, for example, use this method to write the current client to the log.

To use this method, server-side logging must be turned on (see [“Using Server-Side Logging” on page 3-29](#)) and the subroutine `XFPL_LOG` must be entered in the SMC. By default, `XFPL_LOG` is included in the SMC; see [“Using the `xfServerPlus` Application Program Interface” on page 1-30](#).

For example, if your logon routine stored the user name in a string called `username`, you could set it in the log like this:

```
Call LoginObj.setUserString(username)
```

You can then retrieve the string you set. Note that `getUserString()` only gets the string from `setUserString()`; it does not retrieve it from the *x/ServerPlus* log. For example:

```
Dim currUser As String  
currUser = LoginObj.getUserString()
```

Understanding COM Pooling

COM pooling enables you to create a “pool” of connections or COM objects that are active and ready to be used when a client sends a request. You can specify the minimum and maximum pool size, time-out values, and whether objects in the pool should be reused. Depending on the requirements of your application, pooling can significantly improve performance by reducing the time necessary to create objects, establish connections, and perform initialization processing.

Upon starting up, the pool is populated up to the minimum level that you specify. When a client requests an object, the request is satisfied from the objects available in the pool. If no object is available, a new one is created, up to the maximum size of the pool. Once the maximum is reached, requests are queued for a specified length of time.

When the client releases an object, you can specify that it be either returned to the pool or discarded. If it is discarded, causing the number of objects in the pool to drop below the minimum size, a new object will be created.

x/NetLink COM enables you to do two types of pooling: connection pooling and object pooling. You cannot do both simultaneously.

Connection Pooling

Connection pooling creates a pool of *x/ServerPlus* connections. Using connection pooling can reduce the amount of time required to make the *x/NetLink* COM–*x/ServerPlus* connection. The connections in the pool do not time out.

The pooled connections use the host, port, pool return, and other settings for the Default interface in the Synergy Type Library Configuration utility. Once you’ve created a pool of connections, those pooled connections will be used by all Synergy COM objects that request a connection. This means that if you have set a different host and port for a specific interface, those settings will be ignored.

By default, connections are not returned to the pool; to reuse connections, set pool return to true in the Synergy Type Library Configuration utility. See [“Reusing Connections and Objects” on page 14-15](#).

Object Pooling

Object pooling creates a pool of Synergy COM objects. (These are objects instantiated from the procedural interface in your Synergy type library; objects instantiated from structure interfaces are not pooled.) When the pool is created,

the objects are instantiated and connected to *xfServerPlus*. If desired, you can write an initialization method that will be run automatically after the connection is established. In addition, there are several other methods that can be used with pooled objects to perform initialization and cleanup tasks. (See [“Using the Pooling Support Methods”](#) on page 14-22.)

Object pooling is most beneficial when the object is used frequently for a short time and a significant portion of that time is spent acquiring the connection and doing initialization. However, any time there is initialization code that needs to be run, object pooling should improve the performance of your application.

By default, objects are not returned to the pool; stateless objects can be reused by setting pool return to true in the Synergy Type Library Configuration utility or by using the `CanBePooled()` method. See below for more information on reusing objects.

Reusing Connections and Objects

After a connection or object has been used, it can either be returned to the pool for reuse or discarded. By default, all connections and objects are discarded after use. Discarding a connection/object releases resources and ensures that the next client request receives a “clean” connection/object. In general, stateless connections/objects may be returned to the pool, while connections/objects with state (that is, those that persist data) should be discarded after use. If discarding an object causes the pool to fall below the minimum level, a new connection/object is created.

To specify that a *connection* be returned to the pool for re-use, in the Synergy Type Library Configuration utility set the pool return key for the Default interface to “true” (see [page 14-2](#)). Note that the Synergy process that is attached to the connection may have state depending on whether the object that used the connection had state. Because any connection may be used by any object, you may wish to take the precaution of always discarding connections.

To specify that an *object* be returned to the pool, in the Synergy Type Library Configuration utility, set the pool return key to “true” for the interface that instantiates the object (see [page 14-2](#)). You can also determine at runtime if an object should be reused using the `CanBePooled()` method (see [page 14-24](#)).

Implementing Pooling

This section explains how to set up your client machine for connection or object pooling, and how to use the pooling support methods for object pooling.

► To implement connection pooling

For connection pooling, your development process will be essentially the same as it is without pooling. During development, you will need to consider how an object's state may affect your code and whether connections should be returned to the pool or discarded. Then, at deployment time, you'll need to do the following:

1. Set the desired options in the Synergy Type Library Configuration utility. See [“Setting options in the Synergy Type Library Configuration utility” on page 14-17](#).
2. Create and configure a connection pool using the Windows Component Services utility. See [“Creating a pool in Component Services” on page 14-17](#) and [“Configuring the pool” on page 14-19](#).
3. Start the pool. See [“Starting the pool” on page 14-21](#).
4. Run your application as usual.

► To implement object pooling

For object pooling, if you're creating a new application, we recommend that you develop your application right from the start on a machine that supports pooling. This will be much easier than creating the application and then adapting it for pooling later. (Note, however, that if you have an existing *x/NetLink* COM application, you can adapt it for pooling.) You'll need to do the following:

1. (optional, but recommended) Write pooling support methods (activate, initialize, etc.) and add them to the SMC. See [“Using the Pooling Support Methods” on page 14-22](#).
2. Create and register TLBs in the normal manner. If you wrote pooling support methods, include them in the TLB.
3. Set the desired options in the Synergy Type Library Configuration utility. See [“Setting options in the Synergy Type Library Configuration utility” on page 14-17](#).

4. Create and configure an object pool using the Windows Component Services utility. See [“Creating a pool in Component Services” on page 14-17](#) and [“Configuring the pool” on page 14-19](#).
5. Use objects in the normal manner in your client-side code. See [“Programming Tips for Object Pooling” on page 14-25](#).
6. Start the pool, and then run your application. See [“Starting the pool” on page 14-21](#).

Setting Up the Client Machine

To set up the client machine, you need to set some options in the Synergy Type Library Configuration utility. Then, you'll use the Windows Component Services utility to create a COM+ application, add your COM objects to it, configure the pool, and start the pool.

Before starting this procedure, you must install *xy*NetLink COM and register your type libraries. See [“Deploying the Client” on page 14-1](#).

Setting options in the Synergy Type Library Configuration utility

Run the Synergy Type Library Configuration utility and specify the required settings. There are two settings specific to pooling—initialize time-out and pool return—that you may need to set. You must also set *xfhost*, *xfport*, and any others that your application requires.

Connection pooling uses the settings for the Default interface. Object pooling uses the settings for the individual interfaces, if they exist; otherwise the settings for the Default interface are used. See [“Using the Synergy Type Library Configuration Utility” on page 14-2](#) for more information about specifying settings.

For information on changing pool settings see [“Changing the Pool Configuration” on page 14-21](#).

Creating a pool in Component Services

1. From Administrative Tools, select Component Services. (On Vista, run `%windir%\system32\comexp.msc`.)
2. In the tree in the left pane, expand the Component Services node and its subnodes until you see the COM+ Applications node.

Calling Synergy Routines from COM

Implementing Pooling

3. Highlight the COM+ Applications node and select Action > New > Application to start the wizard. On the Welcome screen, click the Next button.
4. Use the wizard to create the application.
 - ▶ On the first screen, create an empty application.
 - ▶ On the second screen, give the application a name and indicate that it is a server application.
 - ▶ On the third screen, select “interactive user” for the account type.
 - ▶ On all platforms except XP, the wizard includes two additional screens regarding application roles. If your application does not use role-based security, you can just use the default role (CreatorOwner) or remove it and specify no role at all. Using the default role means that the user who was signed on when the COM+ application was created will have the necessary authority to start the pool. See your Windows documentation for more information about role-based security.

When you are finished, the new application will display under the COM+ Applications node.

5. Now you are ready to add components to the application. Expand the node for your new application to display the Components subnode.
6. Highlight the Components node and select Action > New > Component to start the wizard. On the Welcome screen, click the Next button.
7. Use the wizard to add components.
 - ▶ On the first screen, indicate that you are going to import components that are already registered.
 - ▶ On the second screen, select components from the list. For connection pooling, select `Xfrerouter.XFConnect.1`. For object pooling, select the specific COM object (it will be named `TLBName.interfaceName.1`). Use CTRL+click to select several components at once.

When you are finished, the newly added components will display under the Components node, as shown in [figure 14-1](#).

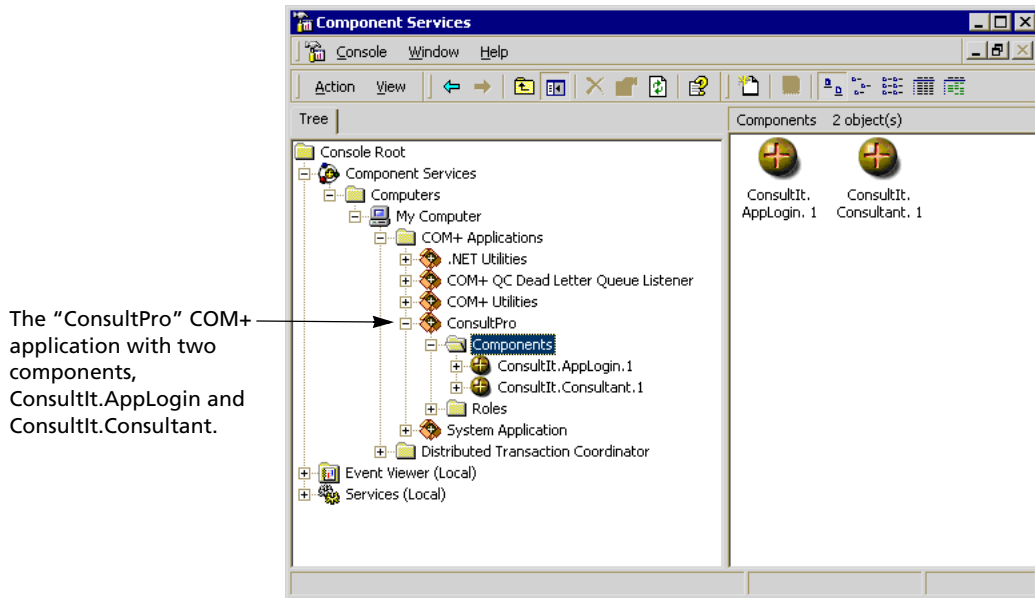


Figure 14-1. The Component Services window, showing a COM+ application with components.

Configuring the pool

1. Highlight the individual component node (e.g., ConsultIt.AppLogin.1 in [figure 14-1](#)) and select Action > Properties.
2. In the Properties dialog box go to the Activation tab. See [figure 14-2](#).
3. Select the "Enable object pooling" checkbox, and then enter values for the following:

Minimum pool size. Enter the number of connections/objects you want created at pool start-up. The pool will never drop below this size.

Maximum pool size. Enter the maximum number of connections/objects that you want in the pool at any one time. When deciding how large to make the pool, keep in mind how many *xfServerPlus* licenses you have available and how many objects you have pooled. The maximum pool size value for all pooled objects should not exceed the number of available *xfServerPlus* licenses.

Creation timeout. Enter the length of time, in milliseconds, that a request should be queued when all connections/objects in the pool are in use and the pool is at its maximum size. If the request is not satisfied within the specified time, an error is thrown.

Calling Synergy Routines from COM

Implementing Pooling

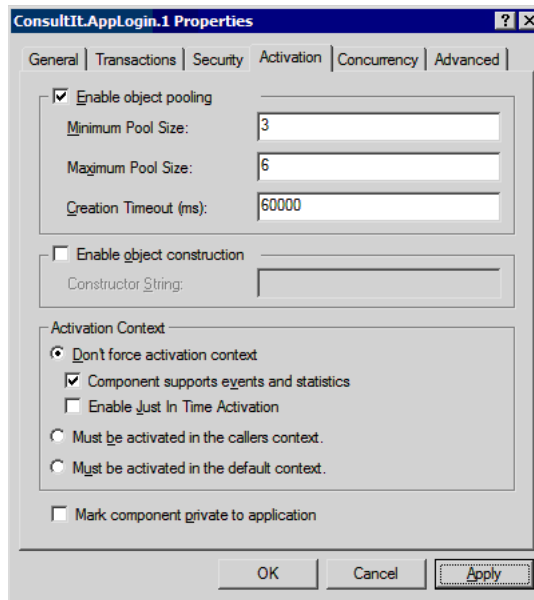


Figure 14-2. Setting properties for a pooled component.

4. Select the “Component supports events and statistics” checkbox. This option enables you to view statistics about the pooled object, such as the pool size and the number of objects in use, in the Component Services window.
5. If the “Enable Just In Time Activation” checkbox is selected, clear it.
6. Click OK in the Properties dialog box. Repeat [steps 1](#) through [6](#) to configure each component.



If you have problems with the pool shutting down unexpectedly, try selecting the “Leave running when idle” option on the Advanced tab of the Properties dialog box that is accessed from the COM+ application node (“ConsultPro” in [figure 14-1](#)). Or, you may want to increase the “Minutes until idle shutdown” value instead. This will cause the pool to wait indefinitely for the first connection, but once a connection to the pool has been made, this timer starts counting. There are numerous other settings that you can configure. The recommended configuration will depend upon your specific application. Refer to your COM+ documentation for details on additional settings.

Starting the pool

1. Highlight the node for the COM+ application you created.
2. Select Action > Start.

Both connection and object pools can also be started automatically. If you do not start the pool manually, it will be populated (i.e., started) the first time a client requests a connection/object. Note, however, that this means that the first user will have to wait for the pool to start and the objects to be initialized. Because the primary goal of pooling is to improve performance at start-up, we recommend manually starting the pool.



(non-XP) If you get a permissions error when attempting to start the pool, it may be related to role-based security. To turn off this security check, right-click on the node for the COM+ application you created and select Properties. In the Properties dialog box, go to the Security tab and clear the "Enforce access checks for this application" box. Click OK in the dialog box.

Changing the Pool Configuration

The pool size and creation time-out values are read when the pool is started. If you change these values in the Component Services utility, you must stop and restart the pool for the changes to take effect. Because restarting the pool will deactivate any objects currently in use, you may want to check the pool status before stopping the pool. To check the status, in Component Services highlight the Components node under your COM+ application node and select View > Status. The number of objects in use displays in the Activated column.

The Synergy TLB Configuration utility settings (xfhost, xfport, pool return, etc.) are read each time a new object is created and added to the pool. Although it is not required that you restart the pool for changes to these settings to take effect, we recommend that you do so.

Using the Pooling Support Methods

NetLink COM enables you to access pooling support methods defined by the COM+ API. These five methods are available for object pooling only—they are not used for connection pooling. The pooling support methods enable you to specify that certain actions be performed at specified times during the object's creation and use. You do not call these methods directly. Rather, they run automatically at various times during the lifetime of the pooled object.

► To use the pooling support methods

1. Write a Synergy routine that performs the desired task. You can name this routine anything you like.
2. Use the Method Definition Utility to add the routine to the SMC. Note the following:
 - In the Method name field, enter the name of the method: Initialize, Activate, Deactivate, CanBePooled, or Cleanup. Because the pooling support methods are called automatically, they must be named correctly.
 - In the Interface name field, specify the interface for the object that this routine applies to.



You can have several routines of the same type if necessary, as long as they are in different interfaces. For example, you might want a separate initialize routine for each of several interfaces. The method name must always be as shown below in “[Pooling support methods](#)”. However, the names of the Synergy routines can vary, so you can include them all in the same ELB if desired.

3. Include the Synergy routine when you build your ELB.
4. Build and register your type library as you normally would.

Pooling support methods

The methods are listed below in the order in which they are called during the object's lifetime.

Initialize method

`status = Initialize()`

`status`—a `^VAL` value that indicates whether the initialization was a success. Returns 0 for success or 1 for failure.

If the return value is 1, the object is destroyed, an error is logged in the `xyNetLink` COM log (if client-side logging is turned on) and the Windows event log, and no other objects in the pool are created. When this happens, the pool will have been started, but there will be no objects in it. To recover from this state, you must stop the pool, fix the `Initialize()` method, rebuild the ELB, and then restart the pool.

`Initialize()` is called when the object is created. Use this method to prepare the environment by opening files, initializing global data, and so forth. Because `Initialize()` is called when the object is created, this method gets called only once per object, even if the object is reused. Compare with `Activate()`.

Activate method

`Activate()`

`Activate()` is called when the object is requested by a client. This method can be used for code that should be executed when the object is actually used.

`Activate()` and `Initialize()` can be used for similar purposes—preparing the environment before using the object. The primary difference between them is that `Activate()` is called *every time* the object is allocated to a client, whereas `Initialize()` is called only *once* when the object is created.

Deactivate method

`Deactivate()`

`Deactivate()` is called when an object is released by the client. It can be used to reset the environment to a known state before an object is returned to the pool. Note that because objects can be reused, this method may be called multiple times. Compare with `Cleanup()`.

CanBePooled method

`status = CanBePooled()`

`status`—a ^VAL value that indicates whether the object should be discarded or reused. Returns 0 if the object should be discarded; returns 1 if the object should be returned to the pool.

The `CanBePooled()` method is called after `Deactivate()`, and can be used to determine at runtime if an object can be re-used. For example, if `Deactivate()` encountered an error, `CanBePooled()` could indicate that the object should be discarded. Or, `Deactivate()` could be written to check how much effort is required to clean up an object before returning it to the pool. If the effort is excessive, and it would be more efficient to discard the object and create a new one, `CanBePooled()` would return 0.

`Canbepooled()` overrides the pool return setting in the Synergy Type Library Configuration utility.

Cleanup method

`Cleanup()`

`Cleanup()` is called by the object's `disconnect()` method. If the object's connection is shared, it is called after the final `disconnect()`. It can be used to do the final object cleanup, such as closing files. This method is called only when objects are going to be discarded (i.e., pool return is set to false or `CanBePooled()` returns 0). If the object is going to be reused, use the `Deactivate()` method to perform cleanup-type activities.

The `Cleanup()` method is also called when socket communication with the client is unexpectedly lost. When the pool is created, the `Cleanup()` method is automatically registered with the `XFPL_REGCLEANUP` routine on the server. (This routine must be in your SMC; see [page 1-34](#) for more information.) Then, if there is a fatal error that causes *xfServerPlus* to lose socket communication with the client, *xfServerPlus* calls the `Cleanup()` routine before shutting down.

Although `Cleanup()` and `Deactivate()` can be used for similar purposes, there are two fundamental differences between them:

- ▶ `Cleanup()` is called only for objects that will be discarded, while `Deactivate()` is called both for objects that will be discarded and for those that will be reused.
- ▶ `Cleanup()` is automatically registered so that *xfServerPlus* can call it when socket communication is lost.

Programming Tips for Object Pooling

Creating poolable objects

To obtain the greatest benefit from object pooling, separate out the initialization and resource acquisition code that is performed for all clients as a prerequisite to actually doing the work of the object. This code can then go in the `Initialize()` method, to be executed when the object is created. Or, depending on the needs of your application, this code may go in the `Activate()` method to be executed when the object is retrieved from the pool by a client.

Writing code that uses pooled objects

In general, ASP or VB code that you write when using pooling will look very similar to code for use without pooling. You will still instantiate objects and call methods. However, because the pooled objects already have a connection to `xjServerPlus`, you do not need to call the `connect()` method. The disconnection from `xjServerPlus` is also handled automatically by the pool, so you do not need to call `disconnect()`. (However, see the next section for why you might sometimes want to.)

For example:

```
Dim LoginObj As ConsultIt.AppLogin
Set LoginObj = CreateObject("ConsultIt.AppLogin")
Dim userID As String
Dim password As String
userID = "MFranklin"
password = "123abc"
Call LoginObj.login(userID, password)
Set LoginObj = Nothing
```



It is important to release the object by setting it to nothing when your program is done with it. If you don't release the object, not only will it continue to use system resources, but its `Deactivate()` method will not be called.

Writing code that will also work when pooling is disabled

If you are planning to use pooling, but also need to be able to run your code when pooling is not enabled (perhaps during development and testing), you'll need to do the following:

- ▶ Include an explicit `connect()` and `disconnect()` in your ASP or VB code as described in [“Using Your Type Library” on page 14-6](#). When you enable COM pooling, the explicit `connect()` and `disconnect()` become superfluous and are ignored.
- ▶ If you are using any of the pooling support methods, you will need to explicitly call those methods when pooling is not enabled. When pooling is enabled, however, your code should *not* call these methods because they will be called automatically. (Unlike `connect()` and `disconnect()`, the pooling support methods are not ignored when pooling is enabled; it is your responsibility to ensure they are not called.)

Method Reference

This is a listing of the utility methods that are included in all procedural interfaces in your Synergy type library.

connect()

`void connect()`

Sends a request to *xfServerPlus* for a dedicated session and connects on the host and port defined in the Windows registry on the client. See [“Connect to *xfServerPlus*” on page 14-8](#).

debugInit()

`void debugInit([in] BSTR client, [out] BSTR* listen_ip,
[out] BSTR* listen_port)`

Starts a connection to *xfServerPlus* so that you can manually connect an *xfServerPlus* session in debug mode. See [“Running an *xfServerPlus* Session in Debug Mode” on page 15-12](#).

client—the IP or name of the *xfNetLink* COM client machine

listen_ip—returns the IP address, in hex, where the client is listening

listen_port—returns the port number where the client is listening

debugStart()

`void debugStart([in] BSTR client)`

Completes the process of connecting in debug mode that was started by `debugInit()`. See [“Running an *xfServerPlus* Session in Debug Mode” on page 15-12](#).

client—the IP or name of the *xfNetLink* COM client machine

disconnect()

`void disconnect()`

Sends a message to close the *xfServerPlus* connection. See [“Disconnect from *xfServerPlus*” on page 14-10](#).

getConnect()

`IUnknown* getConnect()`

Gets the already-established connection of the specified object. See [“Connect to *xfServerPlus*” on page 14-8](#).

getUserString()

BSTR getUserString()

Returns the user string currently stored by setUserString(). See [“Writing to the xfServerPlus Log” on page 14-12.](#)

setUserString()

void setUserString([in] BSTR *userString*)

Passes a user string to the *xfServerPlus* log. See [“Writing to the xfServerPlus Log” on page 14-12.](#)

userString—the text that you want to write to the *xfServerPlus* log

setxfCallTimeout()

void setxfCallTimeout([in] long *seconds*)

Sets the call time-out value, which measures the length of time that the client waits for a return from a remote call. See [“Setting a Call Time-Out” on page 14-12.](#)

seconds—the number of seconds you want the *xfNetLink* COM client to wait for a return from a call to *xfServerPlus*. The default is 1800 seconds (30 minutes).

shareConnect()

void shareConnect([in] IUnknown* *connection*)

Shares the specified connection. See [“Connect to xfServerPlus” on page 14-8.](#)

connection—the object that represents the connection

Error Handling and Troubleshooting in *xfNetLink* COM

This chapter includes a table that lists the errors that can occur when using *xfNetLink* COM Edition and what you can do to resolve them. It also includes information on troubleshooting, including how to turn on logging, run the *xfNetLink* COM test program, and run a debug session.

Handling Errors

xfNetLink COM uses a standard VB or ASP Error object for error handling. In addition to the information in the error message, you can use the standard Error object methods (e.g., `Number()`, `Location()`) to obtain more information about the error.



When you get an error, the log file can also provide information about what went wrong. See [“Using Client-Side Logging” on page 15-9](#). All errors are also written to the Windows application event log, regardless of whether client-side logging is turned on.

The table below lists the errors returned by *xfNetLink* COM, what might have caused the error, and a possible solution.

You may also see *xfServerPlus* status codes returned to the client. Refer to the table on [page 3-16](#).

Error Handling and Troubleshooting in xfNetLink COM

Handling Errors

xfNetLink COM Errors		
Error message	Cause	What to do
Address in use	The port is already being used.	Verify that you specified the correct port for <i>xfServerPlus</i> on both the client and the server and that nothing else is using that port. See “Using the Synergy Type Library Configuration Utility” on page 14-2 and “Running xfServerPlus” on page 3-3.
Address not available	The remote address is not a valid address.	The IP address identifying <i>xfServerPlus</i> is invalid. Verify that you entered it correctly on the client. See “Using the Synergy Type Library Configuration Utility” on page 14-2.
Array index out of bounds	Tried to access an element outside the defined array.	Check the bounds of the array in the type library.
Array size of value is bigger than array size of field	Tried to set the field value to an array that is larger than the defined array size of the field in the repository.	Use a smaller array or increase the size of the array in the repository.
Automation error	Object can't be created.	Look in the log file for the specific error.
Automation error COM+ Activation failed because the activation could not be completed in the specified amount of time.	The pool has reached its maximum size and the client has timed out while waiting for a connection/object to become available.	Increase the creation time-out value or increase the pool size. Both of these values are set in the Properties dialog box for the COM object in Component Services; see page 14-19 .
Cannot coerce type from <i>vartype</i> to VT_DISPATCH	Type coercion to or from an object failed.	Correct your code to use a different data type. See “xfNetLink COM” in Appendix B .
Cannot coerce type from <i>vartype</i> to VT_UNKNOWN		
Cannot coerce type from VT_DISPATCH to <i>vartype</i>		
Cannot coerce type from VT_UNKNOWN to <i>vartype</i>		

Error Handling and Troubleshooting in xfNetLink COM

Handling Errors

xfNetLink COM Errors (continued)		
Error message	Cause	What to do
Class not found in type library	The registered type library does not contain the specified class.	Verify that the type library contains the class and that the type library has been successfully registered.
Connection refused	The host refused the connection.	Verify that the port number is correct on both the client and the server. See “Using the Synergy Type Library Configuration Utility” on page 14-2 and “Running xfServerPlus” on page 3-3 .
Connection timed out	xfNetLink COM timed out because of very slow start-up.	Increase the connect time-out value. If the problem persists, contact your system administrator. See “Using the Synergy Type Library Configuration Utility” on page 14-2 .
Could not create new process	Could not launch db s process.	Ensure that db s.exe exists on the xfServerPlus machine.
DBLDIR not set	The DBLDIR environment variable is not set on the server.	Set DBLDIR on the xfServerPlus machine.
Did not receive expected interface <i>interfaceName</i> for parameter <i>paramNumber</i>	The passed-in structure object doesn't match the structure as specified in the type library (e.g., tried to pass stucture1 when the method is defined to take structure2).	Check your code. Make sure the structure you are passing is the correct one.
Error changing array element type	The vartype of the source array could not be changed into the type of the elements held in the destination array.	Ensure that the type of the elements held in the source array is a type that can be converted to elements of the type held in the destination array as specified in the type library. See “xfNetLink COM” in Appendix B .
Error creating instance of Xfrerouter.XFConnect	xfstructmgr.tlb and/or xfrerouter.dll are not registered properly on the client. If you are using COM pooling, the pool may have failed to start for some reason. For example, xfhost and xfport may not be set. Could also be an internal system error.	Verify that the client is properly set up. See “Deploying the Client” on page 14-1 . If you think it's a pooling issue, check the Windows event log on the web server to see why the pool did not start. Get the Err.Number and Err.Location values from the Error object; save your error log; contact Synergy/DE Developer Support.

Error Handling and Troubleshooting in xfNetLink COM

Handling Errors

xfNetLink COM Errors (continued)		
Error message	Cause	What to do
Error from CoCreateInstance	xfstructmgr.tlb and/or xfrouter.dll are not registered properly on the client. Could also be an internal system error.	Verify that the client is properly set up. See “Deploying the Client” on page 14-1 . Get the Err.Number and Err.Location values from the Error object; save your error log; contact Synergy/DE Developer Support.
Error getting element from SafeArray	There was a problem getting an element from an array.	Get the Err.Number and Err.Location values from the Error object; save your error log; contact Synergy/DE Developer Support.
Error getting progID for CLSID <i>classID</i>	xfstructmgr.tlb and/or xfrouter.dll are not registered properly on the client. Could also be an internal system error.	Verify that the client is properly set up. See “Deploying the Client” on page 14-1 . Verify that the type library containing the class has been successfully registered.
Error loading type library for <i>progID</i>	There was a problem loading the type library for the specified progID.	Verify that the type library has been successfully registered and has not been moved or deleted. Verify that the type library exists for the given progID in the registry. Check the log file for more detail.
Error putting an element of a different type into a SafeArray	Tried to put an element of one type into an array of another type.	Look in the log file for the type of element set and the type expected.
Error putting element into SafeArray	There was a problem putting an element into the array.	Get the Err.Number and Err.Location values from the Error object; save your error log; contact Synergy/DE Developer Support.
Expected a non-array in parameter	Tried to pass an array element into a parameter that didn’t expect an array.	Correct your code to make sure that the parameter you’re passing matches the parameter definition from the type library.
Expected an array in parameter	Tried to pass a non-array parameter into a parameter that expected an array.	Correct your code to make sure that the parameter you’re passing matches the parameter definition from the type library.
Expected IStructManager interface not found	The routine expected to receive a structure parameter, but the object passed was not a structure.	Correct your code to pass a structure object.

Error Handling and Troubleshooting in xfNetLink COM

Handling Errors

xfNetLink COM Errors (continued)		
Error message	Cause	What to do
Field name <i>fieldName</i> already exists in structure <i>structureName</i>	Within the specified structure, there are two fields with the same name.	Edit the repository to remove duplicate field names in the structure, and then update the SMC and regenerate the type library. If you are using alternate field names, check the Alternate name field for duplicates as well.
Host not found	The host name or IP address is defined incorrectly in the registry on the client.	Verify that you entered the correct value for xfhost. See "Using the Synergy Type Library Configuration Utility" on page 14-2.
Host or port not configured	The host or the port have not been set up in the registry on the client.	Set the xfServerPlus host and port on the client. See "Using the Synergy Type Library Configuration Utility" on page 14-2.
Incompatible client version	xfNetLink COM version incompatible with xfServerPlus version.	Upgrade the client or the server to the higher version. Running a newer client with an older server is not a supported configuration.
Incorrect number of parameters supplied to <i>methodName</i>	Tried to make a method call without passing the required number of parameters.	Check the definition of the method to see the required parameters and then correct your code.
Internal System Error	Varies. The detail message gives additional information about the problem.	Get the Err.Number and Err.Location values from the Error object; save your error log; contact Synergy/DE Developer Support.
Invalid connection passed into shareConnect	The object passed to shareConnect() is not a getConnect() object.	Correct your code to use getConnect() as input to shareConnect(). See step 3 on page 14-8 .
Invalid synergy type sent. Structure size passed and SMC size declared don't match.	The size of the parameter in the call doesn't match the size in the SMC. The error message includes the parameter number.	Check the definition of the method to see what size the parameter should be. Correct your code if necessary. If the error is in the SMC, update it and regenerate the type library. If the parameter is a structure, the problem may be that the repository was updated, but the SMC was not. See "Verifying Repository Structure Sizes and Enumerations" on page 2-42.

Error Handling and Troubleshooting in xfNetLink COM

Handling Errors

xfNetLink COM Errors (continued)		
Error message	Cause	What to do
Length of value is greater than the size of the field	Tried to set a field with a value whose length is greater than the defined length of the field in the repository.	Shorten the field value in your code or increase the field length in the repository. If you do the latter, you will also need to update the SMC and regenerate the type library.
Log file name not specified	Client-side logging is turned on but the name of the log file was not set up properly in the registry.	Set logfile in the TLB configuration utility. See “Using the Synergy Type Library Configuration Utility” on page 14-2 .
Login failed	(Win, UNIX) Invalid user name or password used to start xfServerPlus.	Ensure that the user name and password are correct for xfServerPlus. See “Running xfServerPlus” on page 3-3 .
Max port is less than min port	The value for maxport is less than the value for minport.	Ensure that maxport is greater than or equal to minport. See “Using the Synergy Type Library Configuration Utility” on page 14-2 .
Min and max port must be greater than 1024	Either the minport or maxport is set to a value less than or equal to 1024, or only one of these is set.	Ensure that both minport and maxport are set to values greater than 1024. See “Using the Synergy Type Library Configuration Utility” on page 14-2 .
Name not provided for field	A field in a structure does not have a name.	In the repository, ensure that all structure fields are named. If you update the repository structure, you must also update the SMC and regenerate the type library.
No connection established	Tried to call one of the utility methods without first establishing a connection with connect().	Correct your code to use an explicit connect() before calling these methods. See step 3 on page 14-8 .
No more available ports	All ports in the specified range have been used.	Adjust the minport and maxport values to allow for a larger range. See “Using the Synergy Type Library Configuration Utility” on page 14-2 .
No runtime license	There is no Runtime license on the xfServerPlus machine.	Install a Runtime license on the xfServerPlus machine.
Object doesn’t support this property or method	Tried to call a method or access a property that is not supported by the object.	Look at the class in the type library to see the available methods and properties.

Error Handling and Troubleshooting in xfNetLink COM

Handling Errors

xfNetLink COM Errors (continued)		
Error message	Cause	What to do
Parameter direction mismatch	The parameter direction (in, out, or in/out) is defined differently in the SMC than it is in the type library.	Ensure that the parameter is defined correctly in the SMC and then regenerate the type library.
Receive timed out	xfNetLink COM timed out while waiting for a method response from xfServerPlus.	Increase the call time-out value. See “Setting a Call Time-Out” on page 14-12 . If the problem persists, contact your system administrator.
Required input parameter is uninitialized	Tried to pass a required input parameter that is not initialized.	Correct your code to initialize the parameter, or mark the parameter as optional in the SMC and regenerate the type library.
Returned array size does not match size of array in parameter	The array sent back from xfServerPlus does not match the size of the array passed in as a parameter value.	Verify that the size of the array in your client program is the same as the size of the array returned by the Synergy routine.
Send timed out	xfNetLink COM timed out when sending a method call to xfServerPlus.	Increase the call time-out value. See “Setting a Call Time-Out” on page 14-12 . If the problem persists, contact your system administrator.
Size of source array is less than size of destination array	Tried to copy a smaller array to an array that expected a larger one.	Ensure that the array being sent is the same size or larger than the array as it is defined in the type library.
Structure <i>structName</i> size (<i>size</i>) must be greater than or equal to the sum of the selected field sizes (<i>field size sum</i>)	You are using the “Excluded by Web” flag in Repository and the sum of the sizes of the selected fields for this structure (i.e., those not flagged for exclusion) exceeds the size of the structure as determined by Repository.	In your repository, verify which fields you want to select/exclude for this structure. Because the size of the selected fields is too large, you have likely neglected to exclude some fields. See “Passing Structures as Parameters” on page 1-8 . After updating the repository, you will need to update the SMC and regenerate the type library.
Unexpected error	A server-side error occurred.	Note the error number; save your error log; contact Synergy/DE Developer Support.
Uninitialized connection passed into shareConnect	Tried to use shareConnect() without first establishing a connection with connect().	Correct your code to call connect() before shareConnect(). See step 3 on page 14-8 .

Error Handling and Troubleshooting in xfNetLink COM

Handling Errors

xfNetLink COM Errors (continued)		
Error message	Cause	What to do
Uninitialized object for array element <i>x</i>	Tried to pass an array of structures as a parameter value, and one of the structures (element <i>x</i>) in the array was not created using <code>CreateObject()</code> .	Instantiate all elements of the array with <code>CreateObject()</code> before passing the array in a method call.
Uninitialized object in parameter <i>paramNumber</i>	Tried to pass an object as a parameter value without first initializing it with <code>CreateObject()</code> .	Instantiate all objects with <code>CreateObject()</code> before passing them as parameter values. See "Using Your Type Library" on page 14-6 for more information.
Unknown Windows error	Windows API failed.	Check the <code>Err.Location</code> value in the Error object to see where the error occurred.
<i>xfServerPlus</i> license count exceeded	<i>xfServerPlus</i> has exceeded the number of available licenses.	Upgrade the number of licenses.
<i>xfServerPlus</i> license expired	The 14-day demo period or an extended demo period has expired.	Contact your Synergy/DE customer service representative.
<i>xfServerPlus</i> not licensed	There is no license for <i>xfServerPlus</i> .	Purchase a license.
<i>xfServerPlus</i> not supported by the running server	<i>xfServerPlus</i> is not running on the specified port.	Verify that <i>xfServerPlus</i> is running on the default port or on the port you specified when configuring the TLB with the Synergy Type Library Configuration utility.
<i>xfServerPlus</i> version not supported	The version of <i>xfServerPlus</i> is not compatible with the version of <i>xfNetLink</i> for the current operation.	Upgrade <i>xfServerPlus</i> to the most current version.

Troubleshooting Techniques

Error messages don't always provide enough diagnostic information to solve a problem. In such cases, you can take advantage of the additional debugging options provided with *xfNetLink* and *xfServerPlus*: client-side logging, the *xfNetLink* COM test program, and the ability to run an *xfServerPlus* session in debug mode.

You may also want to view the server-side logs (see [“Using Server-Side Logging” on page 3-29](#)) and run the test program included with *xfServerPlus* (see [“Testing xfServerPlus” on page 3-15](#)).

Using Client-Side Logging

You may want to use client-side logging when access to the server-side log is inconvenient, or when you have used server-side logs and seen differences between what the client program is passing and what the server log shows is being received. All *xfNetLink* COM client errors are always written to the Windows application event log, regardless of whether logging is turned on. Client-side logging, however, enables you to log other types of information (see below).

To turn on client-side logging, set the logging and logfile values in the Synergy Type Library Configuration utility. See [“Using the Synergy Type Library Configuration Utility” on page 14-2](#) for details on setting these values.

Depending on what level of logging you select, the log may include the following information:

- ▶ The instance identifier of the object writing to the log. Each line in the log file is preceded by this identifier. It can be helpful when there are several objects writing to the same log file.
- ▶ Errors
- ▶ Method name
- ▶ Parameter names, types, and values before the call
- ▶ Type coercions that take place
- ▶ Parameter names, types, and values after the call
- ▶ Return values
- ▶ Class ID and associated type library
- ▶ Interface ID
- ▶ Packets sent from and received by the client

Error Handling and Troubleshooting in xfNetLink COM

Troubleshooting Techniques

For the sample below, we turned on logging for errors, user method calls, interface creation/query calls, and packets in the Synergy TLB Configuration utility. “11c3b30” is the identifier of the object writing to this log; all objects write to a single log file.

```
11c3b30:Successfully loaded type library
C:\xfNLCOM\Examples\xfTest.tlb
11c3b30:created instance of class name=xfTest
clsid={09938ABB-61AB-11D4-8195-0050DA5DC5B1}
11c3b30:initial interface name=IxfTest
iid={09938A92-61AB-11D4-8195-0050DA5DC5B1}
11c3b30:connecting to xfServerPlus on host=elmo port=2356
11c3b30:bound to local port 3385 on host tiger (111.22.33.44)
*****
11c3b30:Invoking function_ten
11c3b30:initial parameter values: # parameters:2
11c3b30:parameter 1 name=p1 dir=in vartype=VT_BYREF|VT_BSTR
value="abcde"
11c3b30:parameter 2 name=p2 dir=inout vartype=VT_BYREF|VT_BSTR
value="54321"
11c3b30:calling user method function_ten
11c3b30:sending method call packet
11c3b30:packet length=37
11c3b30:% Jxfpl_tst6;2;AL5#abcde;AL5#54321;
11c3b30:received method call reply packet
11c3b30:packet length=48
11c3b30:0 Rxfpl_tst6;002;000DE9#123456789;002AL4#back;
11c3b30:done calling user method function_ten
11c3b30:after call: # parameters:2
11c3b30:parameter 1 name=p1 dir=in vartype=VT_BYREF|VT_BSTR
value="abcde"
11c3b30:parameter 2 name=p2 dir=inout vartype=VT_BYREF|VT_BSTR
value="back"
11c3b30:returning vartype=VT_BSTR value="123456789"
```

Testing *xfNetLink COM*

The **xfTestchm.exe** program, distributed with *xfNetLink COM*, can help you determine if your system is set up and working properly. The **xfTestchm** files are installed in the *xfNLCOM\Examples* directory. **xfTestchm** is a Visual Basic program and requires that you have the VB runtime installed. If you don't have the VB runtime, you can download it from the Microsoft web site.

xfTestchm runs several tests, which send different types of data back and forth between the COM client and the Synergy server. This program makes calls to a test ELB or shared image named **xfpl_tst**, which is distributed with *xfServerPlus*. There are entries in the SMC for use by the test program. (These are the methods in the *xfTest* interface in the distributed SMC.) If the ELB or any of the methods are missing, the tests will fail.



If the methods in the *xfTest* interface are not present in your SMC, you can import them from the **defaultsmc.xml** file. See [“Importing and Exporting Methods” on page 2-39](#).

There is also a test program for *xfServerPlus*; see [“Testing *xfServerPlus*” on page 3-15](#).

► To run the *xfNetLink COM* test program

1. Make sure *xfServerPlus* has been started on the server machine.
2. Make sure *xfNetLink COM* has been installed on the client machine.
3. Start the Synergy Type Library Configuration utility and do the following:
 - Click the Add TLB button and add **xfTest.tlb**. It is located in the *xfNLCOM\Examples* directory.
 - Specify settings for *xfport* and *xfhost* for **xfTest.tlb**. We recommend that you turn on logging and specify a log file, so that if there are any errors they will be written to the log. If you need help using this utility, see [“Using the Synergy Type Library Configuration Utility” on page 14-2](#).
4. Run **xfTestchm.exe** from the *xfNLCOM\Examples* directory.
5. If you don't want to use a shared connection, uncheck the box. A shared connection will run faster.
6. Click the Run Tests button.

If the tests run successfully, you'll see a success message. If any tests are unsuccessful, an error message will display on the screen. The error message text should help you determine what needs to be done to correct the problem. Refer to the error message table beginning on [page 15-2](#). If you did not turn on logging, you may want to do so and run the test again. Then, if you cannot solve the problem, save the log file and contact Synergy/DE Developer Support.

Running an xfServerPlus Session in Debug Mode

During normal operation, *xfServerPlus* runs as a background process without support for console operations, complex user interfaces, or debugging. This improves efficiency and minimizes memory requirements. However, there may be times when you need to run the debugger on Synergy code in the ELBs that are being called from *xfServerPlus*. By manually connecting an *xfServerPlus* session to your COM client application, you can run your Synergy server routines in debug mode so that you can uncover problems that are showing up as errors in your distributed application.



We recommend that you use the Telnet method for debugging if the operating system of your *xfServerPlus* machine is Windows or UNIX. See ["Debugging Remote Synergy Routines via Telnet"](#) on [page 3-42](#) for instructions.

Running in debug mode on Windows and UNIX

Use this procedure if the operating system of your *xfServerPlus* machine is Windows or UNIX.

If your SMC files or *xfpl.ini* file are not in the default location (DBLDIR), you will need to either move them to DBLDIR or set XFPL_SMCPATH and XFPL_INIPATH *in the environment* to point to the location of the files before starting *xfpl.dbr* ([step 4](#)). (Note: When XFPL_SMCPATH and XFPL_INIPATH are set in the registry or *synrc*, they are read by *rsynd*. Since *rsynd* is bypassed when you run in debug mode, the registry/*synrc* settings do not get read.)



If your client application uses Active Server Pages, you will need to use two ASP pages to run in debug mode. Use the first page to make the call to `debugInit()` and return the HTML that displays the IP and port. Then, use a second page to make the `debugStart()` call. After submitting the second page, the client will go into wait mode while you start *xfServerPlus*.

1. Use the `debugInit()` method to initiate a debug session. You pass the name of the client machine as a parameter. This method binds an IP address and port number for listening, and then returns the IP and port. You need to include code that displays the IP address (in hex) and port on the screen.

For example, in VB:

```
; declare variables
Dim LoginObj As Object
Dim listen_ip As String
Dim listen_port As String
; instantiate object and make call
Set LoginObj = CreateObject("ConsultIt.AppLogin")
Call LoginObj.debugInit("myMachine", listen_ip, listen_port)
; display hex IP and port
MsgBox ("IP = "& listen_ip & " Port = "& listen_port)
```

2. When the IP and port display on the screen, write them down. You'll need them in [step 4](#). For example:

```
IP = 6F16212C
Port = 1082
```



Once the IP address and port display on the screen, you have a limited amount of time in which to manually start *xfServerPlus* in debug mode on the server machine, specify a breakpoint, and type "go". The default time-out for debug mode is 10 minutes. You can change this value by setting the connect time-out value in the Synergy TLB Configuration utility. See ["Using the Synergy Type Library Configuration Utility" on page 14-2](#).

3. Use `debugStart()` to complete the connection process:

```
Call LoginObj.debugStart("myMachine")
```

At this point, the client application has opened a socket and is waiting for the server to call it back.

4. Go to the machine running *xfServerPlus*, start **xfpl.dbr**, and pass the IP and port to *xfServerPlus*. Type the alpha characters in the IP address in uppercase.

```
dbr -d xfpl hexadecimal_ip listen_port
```

For example:

```
dbr -d xfpl 6F16212C 1082
```

xfServerPlus starts up in the debugger window.

Error Handling and Troubleshooting in xfNetLink COM

Troubleshooting Techniques

5. Set an initial breakpoint in the **xfpl** program at the **XFPL_DEBUG_BREAK** routine. In the debugger enter

```
break xfpl_debug_break
```

and then enter

```
go
```

xfServerPlus is now connected to the client on the specified port. The server waits while the client program resumes and makes its first call. The program will then break at the **XFPL_DEBUG_BREAK** routine. This breakpoint occurs just after *xfServerPlus* has opened the ELB for the first method called by your application. (Note that any ELBs linked to this ELB will also be opened.) The ELB must be opened before you can set breakpoints in the routines within it.

6. If the Synergy routine you need to debug is in one of the opened ELBs, just specify a breakpoint in that routine. If the routine you want to debug is in a different (unopened) ELB, use the **OPENELB** debugger command to open that ELB. (You can also continue running your client application until the ELB is opened by *xfServerPlus*. However, because you set a breakpoint at **XFPL_DEBUG_BREAK**, it will break at each method call, so using the **OPENELB** command is more efficient.)



For general information about the Synergy debugger, see the “[Debugging Your Synergy Programs](#)” chapter in *Synergy Language Tools*. For details on the **OPENELB** command, see [OPENELB](#) in that same chapter.

Running in debug mode on OpenVMS

Use this procedure if the operating system of your *xfServerPlus* machine is OpenVMS.

1. Make sure *xfServerPlus* is running on an unused port. If necessary, restart it to ensure that it's using an unused port.
2. On the machine running *xfServerPlus*, enter

```
$ run DBLDIR:xfpld
```

You'll see output similar to the following:

```
*****
***  DEBUG 9.3.1    ***
BREAK AT 152 IN XFPL (LAUNCHER.DBL;6) ON ENTRY
%DBG-E-Could not open source file "LAUNCHER.DBL;1"
Dbg!Dbg>
*****
```



If you have created shared image logicals for the shared images used by *xfServerPlus*, you can skip [step 3](#). Instead, set a breakpoint for your shared image and routine as described in [step 6](#). You'll then be prompted for the port number ([step 4](#)). Once you start your client program ([step 5](#)), the debug session will break at the breakpoint you set.

3. Set an initial breakpoint in the *xfpl* program at the `XFPL_DEBUG_BREAK` routine. In the debugger, enter

```
break xfpl_debug_break
```

and then enter

```
go
```

4. When prompted, enter the port number that *xfServerPlus* is running on (from [step 1](#)).
5. Start your client application in the usual manner. After *xfNetLink* connects, the debug session will break at the `XFPL_DEBUG_BREAK` routine.
6. Set a breakpoint for your Synergy shared image and routine:

```
set break image/routine
```

(For details, see [BREAK](#) in the “Debugging Your Synergy Programs” chapter of *Synergy Language Tools*.)

Note that if you set a breakpoint at `XFPL_DEBUG_BREAK`, the debugger will break at `XFPL_DEBUG_BREAK` for each method call your client makes.



Although you do not need to use the `OPENELB` debugger command before setting the first breakpoint in your shared image, you may need to use it if your code does an `XSUBR` or `RCB_SETFNC` without specifying a shared image. For details on the `OPENELB` command, see [OPENELB](#) in the “Debugging Your Synergy Programs” chapter of *Synergy Language Tools*.

Part V: *xf*NetLink .NET Edition

Creating Synergy .NET Assemblies

This chapter gives an overview of the tasks you must perform to set up and use *xyNetLink* .NET Edition. It explains how to generate C# classes from SMC and repository definitions and build an assembly for your Synergy methods. The generated C# classes use the *xyNetLink* .NET classes internally to connect to *xyServerPlus*. The assembly that you build can be used in any .NET client application to remotely access your Synergy business logic on the *xyServerPlus* machine.

System Requirements

To build a distributed system with *xyNetLink* .NET and *xyServerPlus*, you'll need the items listed below in addition to *xyNetLink* and *xyServerPlus*.

- ▶ Microsoft Visual Studio version 2005 or higher
- ▶ The .NET Framework SDK version 2.0 or higher (installed with Visual Studio, or you can download it from Microsoft's web site)
- ▶ Web server, if you plan to develop a web application

You'll also need to be familiar with Microsoft's .NET Framework and Visual Studio. Here are some of the topics you may need to research:

- ▶ Public vs. private deployment of assemblies
- ▶ Versioning
- ▶ Signing
- ▶ Public and private keys
- ▶ Pooling
- ▶ The GAC (global assembly cache)

System Overview

Figure 16-1 shows the primary components of a distributed application that accesses Synergy code from a .NET client. The diagram describes two machines:

- ▶ A client machine running *x/NetLink* .NET, the .NET Framework, and an application that uses a Synergy assembly. If you're developing a two-tier system with a Windows client application, the client is the end-user's machine. If you're developing a three-tier system with a web client application, the client is the web server machine.
- ▶ A Synergy server running *x/ServerPlus*, which handles the remote execution of Synergy routines. The routines are made available for remote execution by including them in an ELB or shared image and defining them in the Synergy Method Catalog (SMC), also located on the server machine. You can populate the SMC with routine information by entering it manually through the Method Definition Utility or by attributing your code, running *dbl2xml* to create an XML file, and then loading that file into the SMC. You may use multiple servers; each machine requires an *x/ServerPlus* license.

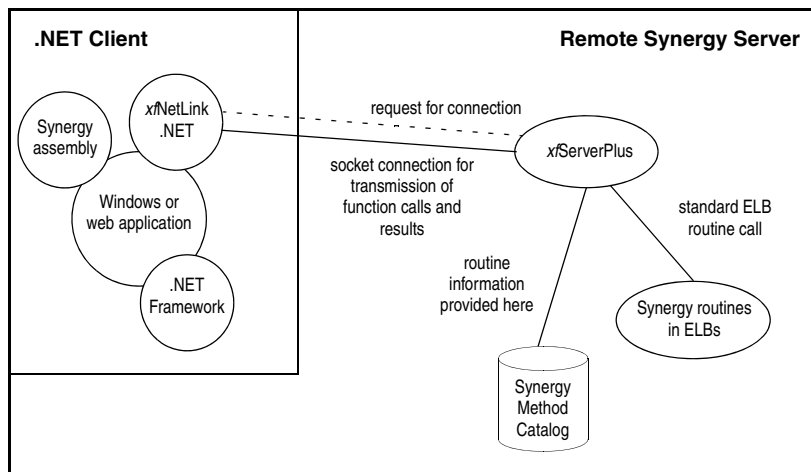


Figure 16-1. Accessing Synergy from .NET.

x/NetLink .NET enables you to use your existing Synergy code without rewriting it, provided that the code is already written in the form of an external subroutine or function. If any of the routines require input from or send messages to the user, or if they might generate untrapped errors, they must be adjusted to work as server-level logic.

The Big Picture

This section lists all the steps that need to be completed to successfully create a distributed system using *x/NetLink* .NET. Note that these steps may not all be done by you, the Synergy developer. For example, you may create the Synergy assembly and give it to a .NET developer to create a web front-end for your application. The .NET developer may also work with a web-page developer to create the HTML portions of the user interface. However, this section should give you a feel for everything that needs to be done, regardless of who does it.

1. Install Visual Studio and the .NET Framework SDK.
2. Install the necessary Synergy software. The components you need to install vary depending on your set-up (e.g., what OS your source files are on, where you intend to do the development, and so on). Note the following:
 - ▶ The tools required to generate the C# classes from your SMC and repository definitions are part of *x/NetLink* .NET and must be installed on a Windows system.
 - ▶ The Professional Series Development Environment (or Workbench) installation includes the SMC/ELB comparison utility, the MDU and SMC files, **genxml**, and the test skeleton generation utility. On UNIX and OpenVMS, these utilities are part of the Synergy Language installation.
 - ▶ *x/ServerPlus* must be installed on your Synergy server machine. The *x/ServerPlus* installation also includes the MDU, the SMC files, and **genxml**. For detailed steps on setting up *x/ServerPlus*, see [“The Big Picture” on page 3-1](#).
3. Modularize your existing Synergy code for the routines that you want to access remotely and encapsulate them in ELBs or shared images. See [chapter 1](#), [“Preparing Your Synergy Server Code.”](#)
4. Populate the Synergy Method Catalog with information about your Synergy routines. You can do this by attributing your code or by entering data with the Method Definition Utility. As you do this, you’ll group the routines into

interfaces. If you choose to put the SMC somewhere other than DBLDIR, set the XFPL_SMCPATH environment variable. See [chapter 2, “Defining Your Synergy Methods.”](#)

5. In the **xfpl.ini** file, set logging options for the *xfServerPlus* log and set logicals that point to the ELBs you specified in the SMC (if necessary). You may also need to set other options in the **xfpl.ini** file; see [Appendix A](#) for a complete list of **xfpl.ini** configuration settings.

If you choose to put the **xfpl.ini** file somewhere other than DBLDIR, set the XFPL_INIPATH environment variable.

See [chapter 3, “Configuring and Running xfServerPlus,”](#) for information on the *xfServerPlus* log and XFPL_INIPATH. See “[Defining Logicals](#)” on [page 1-4](#) for information on setting logicals that point to your ELBs.

6. Create a user account on the *xfServerPlus* machine to run *xfServerPlus* sessions and then start *xfServerPlus*. See “[Running xfServerPlus](#)” on [page 3-3](#).
7. If you’re using Workbench, create a Synergy/DE .NET Component project and add it to the workspace. Specify the component information: a name and location for the assembly, the location of the SMC, the location of repository files (if applicable), the namespace and key file, the interfaces you want to include, and (if desired) alternate names for those interfaces. See “[Creating a Synergy/DE .NET Component Project](#)” on [page 16-6](#).
8. If you’re using Workbench, use the menu option to generate the C# classes. See “[Generating C# Classes](#)” on [page 16-10](#).

If you’re not using Workbench, run the **genxml.dbr** utility to create a Synergy XML file, and then run **gencs.exe** to generate the C# classes. See “[Creating an Assembly from the Command Line](#)” on [page 16-13](#).

9. (optional) Modify the generated C# source files and the metadata for the assembly as necessary. See “[Editing the Generated Files](#)” on [page 16-26](#).
10. (optional) Set SYNCSCOPT to include C# compiler options that you want added to the command line when the assembly is built. See **SYNCSCOPT** in the “[Environment Variables](#)” chapter of the *Environment Variables and System Options* manual.

11. If you're using Workbench, use the menu option to compile the C# classes and create an assembly. See [“Building the Assembly” on page 16-11](#).

If you're not using Workbench, run the batch file that was created by **gens** to compile the C# files and build the Synergy assembly. See [“Building the Assembly” on page 16-20](#).

12. (optional) Generate the API documentation for the methods in your assembly using a third-party utility. See [“Generating API Documentation” on page 16-27](#).
13. Distribute the assembly and other necessary files to the .NET developer, who will then use the assembly when writing the client-side code for your distributed application. See [“Deploying the Client \(for Development\)” on page 17-1](#) and [“Using Your Synergy .NET Assembly” on page 17-9](#).

Creating an Assembly in Workbench

The component generation tools enable you to create a .NET assembly of Synergy methods. The assembly can be used in any .NET client application to make remote calls to Synergy routines. To create an assembly, you must do the following:

- ▶ Create a Synergy/DE .NET Component Project in Workbench and specify information about how the assembly should be constructed.
- ▶ Generate the C# classes.
- ▶ Edit the C# code, if necessary.
- ▶ Build the assembly.



You can generate C# classes and build the assembly from the command line rather than from Workbench. If your Synergy code is on UNIX or OpenVMS, you can use the command-line tools to generate the XML file there, and then move the file to Windows to create the assembly. See [“Creating an Assembly from the Command Line” on page 16-13](#).

Creating a Synergy/DE .NET Component Project

1. In Workbench, select Project > New, and then select Synergy/DE .NET Component from the list of project types. (Expand the Synergy/DE node in the Project type display to select Synergy/DE .NET Component.)
2. Give the project a name in the Project name field, and indicate whether you want to create a new workspace or add the project to an existing workspace. For more information on using this dialog, as well as information about basic Workbench features, see the Workbench online help or the “[Developing Your Application in Workbench](#)” chapter of *Getting Started with Synergy/DE*.

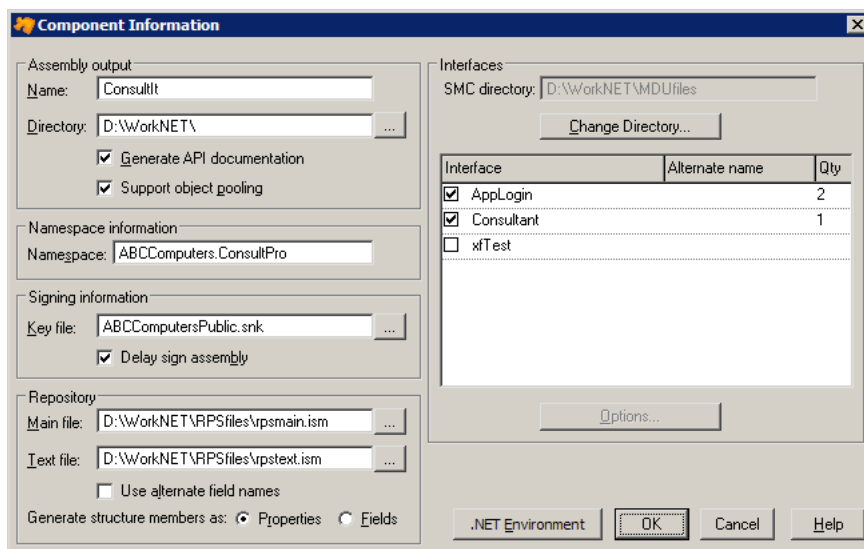


Figure 16-2. The Component Information dialog box for a Synergy/DE .NET Component project.

3. Specify the following information in the Component Information dialog box. (This dialog displays automatically when you create a new .NET Component project. If you need to display it later, select Build > Component Information.)
Name. Enter a name for the assembly. The default is the project name.

Directory. Specify the directory in which to create the class files and the assembly. The default is the project location. If you enter a logical in this field, it must be followed by a colon (e.g., DBLDIR:). When you generate classes, the utility will create a subdirectory within this directory, named with the assembly name, and put all the generated files in it.

Generate API documentation. Indicate whether you want to create an XML file that can be used to generate API documentation. In order to have useful documentation, you must add comments for your methods and parameters. The XML file is created when you build the assembly. See [“Generating API Documentation” on page 16-27](#) for more information.

Support object pooling. Indicate whether you want to enable the classes in this assembly to be pooled.



Selecting the “Support object pooling” option causes the generated procedural classes to be derived from the `ServicedComponent` class. It also changes the status of some methods from public to private or protected, and causes the `getConnection()` and `shareConnect()` methods to be excluded from the procedural classes. Thus, you should use this option only when you are certain the object will be pooled. See [“Understanding .NET Pooling” on page 17-26](#) and [“Implementing Pooling” on page 17-28](#) for more information on pooling.

Namespace. (optional) Specify a namespace for the assembly. All classes generated for the assembly will use this namespace. The namespace is used to ensure that each class is unique. Microsoft recommends that namespaces use the format *CompanyName.ProductName* (e.g., ABCComputers.ConsultPro). The namespace is appended to the beginning of the class name (e.g., ABCComputers.ConsultPro.MyClass). If you do not specify a namespace, the default namespace *assembly_nameNS* will be used.

Key file. (optional) Specify the full path and filename of the strong name key file that will be used to strong name the assembly. You must create the file using Microsoft’s `sn.exe` utility and place it in the desired location before building the assembly. `xfNetLink .NET` will verify that the file exists. See [“Using Your Own Key File” on page 16-25](#) for more information.

If you do not supply a key file name, `xfNetLink .NET` will generate public and private keys in a strong name key file named with the assembly name.

Delay sign assembly. Indicate whether you want to delay sign the assembly. This option is valid only when you use your own key file. See [“Using Your Own Key File” on page 16-25](#) for more information.

Repository main file. (optional) If any of the methods that will be included in this assembly pass structures as parameters, specify the location of the repository main file for those structures. This must match the repository that was used when entering data in the SMC. If you specify a main file, you must also specify a text file.

The default is the value of the environment variable RPSMFIL. If it is not defined, the default is **RPSDAT:rpsmain.ism**. If neither RPSMFIL nor RPSDAT are defined, the default is **rpsmain.ism** in the path specified in the Working directory property of the project. If the Working directory is not defined, the default is **rpsmain.ism** in the location where the project is stored.

Repository text file. (optional) If any of the methods that will be included in this assembly pass structures as parameters, specify the location of the repository text file for those structures. This must match the repository that was used when entering data in the SMC. If you specify a text file, you must also specify a main file.

The default is the value of the environment variable RPSTFIL. If it is not defined, the default is **RPSDAT:rpstext.ism**. If neither RPSTFIL nor RPSDAT are defined, the default is **rpstext.ism** in the path specified in the Working directory property of the project. If the Working directory is not defined, the default is **rpstext.ism** in the location where the project is stored.

Use alternate field names. Indicate whether you want to use the value in the Alternate name field in Repository instead of the value in the Name field as the property or field name. If not selected, the field name in the structure becomes the property/field name in the C# class. If selected, the value in the Alternate name field is used if it exists; else, the value in the Name field is used.

Generate structure members as. Indicate whether you want to generate structure members as properties with private fields or as public fields. Which you choose depends on your application. If you are planning to use .NET controls, generate properties; the properties have “get” and “set” methods, which can be assigned to .NET controls in Visual Studio. You must also generate properties to take advantage of Repository’s read-only flag and the class Changed property.

If you are not planning to use .NET controls, and don't need the read-only flag or Changed property, you can generate either fields or properties; using fields may improve performance.

SMC directory. This field displays the path for the Synergy Method Catalog that this assembly uses. If set, Workbench will use XFPL_SMCPATH to find the SMC files; else, the default is DBLDIR. To change the SMC directory, click the Change Directory button. In the Browse for SMC Directory dialog box, navigate to the directory that contains the SMC, double-click to select it, and click OK. The selected path will display in the SMC directory field and the list of interfaces will be refreshed, displaying all interfaces in the selected SMC.

Interfaces. Select the interfaces you want to include in the assembly by clicking in the box to the left of the interface name. A C# class, named with the interface name, will be generated for each selected interface.

Alternate name. (optional) You can provide an alternate name for any interface you select. To specify an alternate name, highlight a selected interface, and then click the Options button to display the Interface Options dialog. Enter another name in the Alternate interface name field and click OK.

You may wish to use an alternate name if the interface name in the SMC is not what you want users to see as the class name. In addition, if your interface names differ only in case, or if you have a structure with the same name as an interface, you can use the alternate interface name to avoid having numbers appended to the class names. See the note on [page 2-25](#).

To remove an alternate interface name, clear the checkbox for the interface and then reselect it. This action will also reset the quantity to 1.

Qty. By default, a single class is generated for each selected interface; however, you can choose to generate multiple classes for any of the interfaces you select. (See “Using Multiple Copies of the Same Class” on [page 17-12](#) for why you might want to do this.) To generate multiple classes, select (and highlight) the interface, and then click the Options button to display the Interface Options dialog. Specify the *total* number of classes you want to generate in the “Total number of classes to generate for this interface” field and click OK. Valid values are 1 through 999.

The first instance of the class will be named as usual with the interface name (or alternate name, if specified). Subsequent instances will be named with the interface name followed by a number, which will increment.

4. Click OK in the Component Information dialog box.

Controlling the .NET Environment

By default, Workbench uses the batch file `%VS $m$ COMNTOOLS%vsvars32.bat` (where m is the highest version of Visual Studio that is installed on the system) to set the environment variables used by .NET. You can use a different `vsvars32.bat` file or use `sdkvars.bat` or `vcvarsall.bat`. (Note that you cannot run `vcvarsall.bat` with parameters from Workbench as you can from the command line; the parameters are required for 64-bit. See the note on [page 16-20](#).)

The default target version for the generated assembly is Framework 2.0. This is the version of the .NET Framework that will be required to run the generated assembly. You can change this to 3.5 or 4.0.

1. Open the Component Information dialog and click the .NET Environment button.
2. To specify a different batch file, enter it in the “.NET environment batch file” field or click the browse button and select it.
3. To change the target version for the generated assembly, select it from the drop-down list. If you select a version that you do not have installed, the default is used.
4. Click OK.

Generating C# Classes

1. To generate the C# classes, select Build > Generate C# Classes in Workbench. This command will do the following:
 - ▶ Run the SMC/ELB Comparison utility (`smc_elb.exe`; see [page 2-55](#) for more information).
 - ▶ Create a C# source file for each interface you selected in the Component Information dialog box and for each structure and group within the selected interfaces. There will also be a C# interface file generated for each selected interface. See “[Understanding the Generated Classes](#)” on [page 16-22](#) for more information on these files.
 - ▶ Create a batch file to be used later to compile the classes, generate the assembly, and create the XML file for the API documentation.
 - ▶ Create a response file named *assembly_name.rsp*, which is used by the batch file to compile the classes.

- ▶ Create a file named **AssemblyInfo.cs**, which contains information about the assembly. You can customize this file; see [“Editing Information in AssemblyInfo.cs” on page 16-26](#).
- ▶ Add the C# source files, the C# interface files, and the **AssemblyInfo.cs** file to the project. You can access these files from the Projects tab or the Symbols tab in the project toolbar. See [figure 16-3 on page 16-12](#).

If you have previously generated classes for this project, you will be prompted to overwrite them. If you're regenerating classes for the *same interfaces*, the .cs files will be overwritten and any changes you made to them will be lost. Note that the **AssemblyInfo.cs** file is *not* overwritten.

2. Edit the files as necessary before building the assembly. See [“Editing the Generated Files” on page 16-26](#).

Building the Assembly

After you've made any necessary edits to the C# source files and the **AssemblyInfo.cs** file, you are ready to build the assembly.

If desired, you can pass C# compiler commands to the command line that Workbench uses to build the assembly. To do this, set the SYNCSCOPT environment variable to any valid C# compiler command either from the Workbench command line or on the Open tab of the Project Properties dialog. See [SYNCSCOPT](#) in the “Environment Variables” chapter of the *Environment Variables and System Options* manual for more information.

1. Verify that the .NET Component project is the active project.
2. (optional) Set SYNCSCOPT.
3. Select Build > Build Assembly. This command will do the following:
 - ▶ Compile the .cs files.
 - ▶ Build the assembly.
 - ▶ Generate an XML file named *assembly_nameAPI.xml* if the “Generate API documentation” flag was set in the Component Information dialog box when you created the classes. See [“Generating API Documentation” on page 16-27](#) for more information on what to do with this XML file.

Creating Synergy .NET Assemblies

Creating an Assembly in Workbench

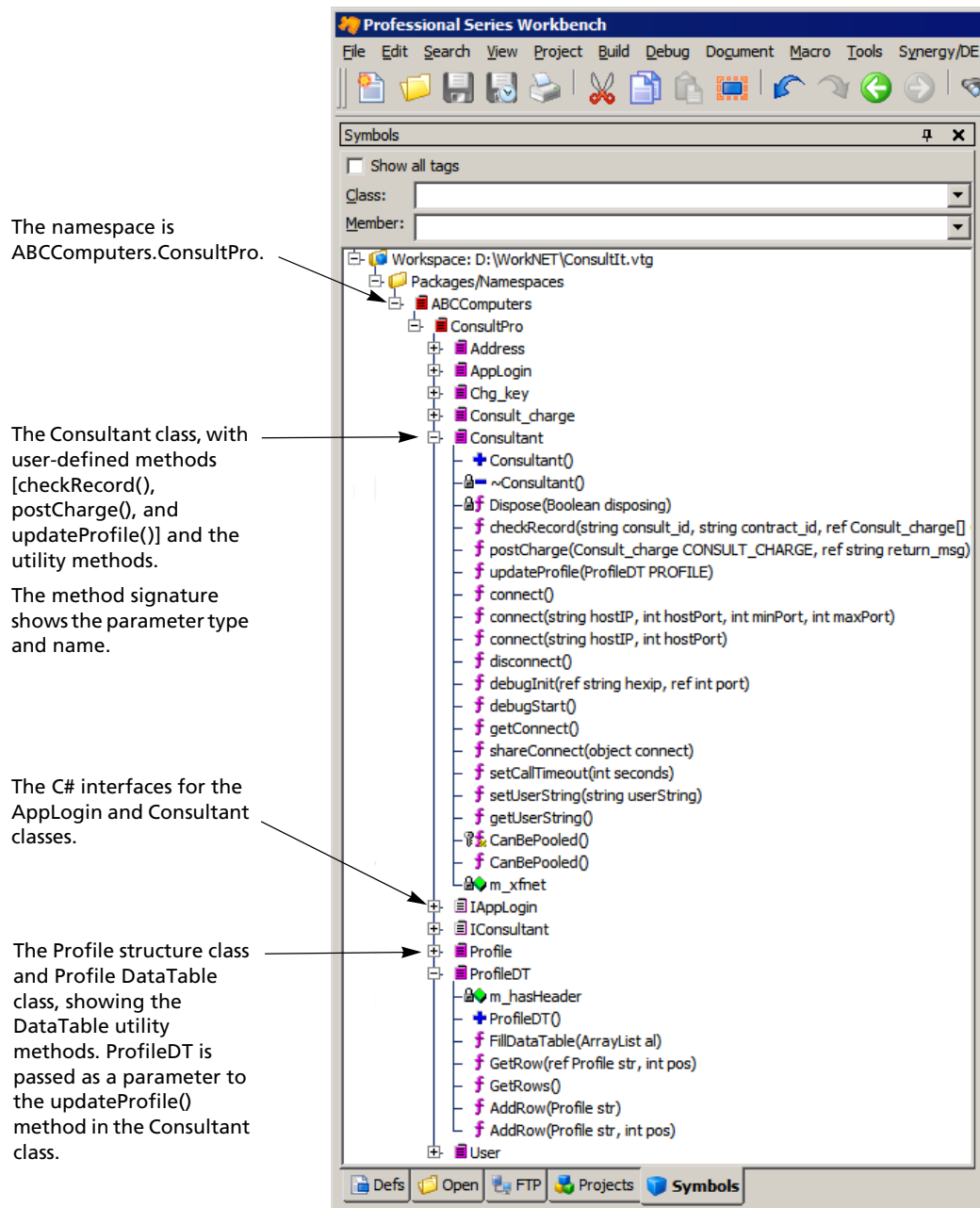


Figure 16-3. Viewing the generated C# classes and C# interfaces in Workbench.

Creating an Assembly from the Command Line

Follow the instructions in this section to generate C# classes and build a Synergy assembly from the command line. You will need to do the following:

1. Use the **genxml** utility to create a Synergy XML file. See [“The genxml Utility”](#) below.
2. Use the **gens** utility to generate the C# classes from the XML file. See [“The gens Utility” on page 16-16](#).
3. (optional) Edit the source files. See [“Editing the Generated Files” on page 16-26](#).
4. Run the batch file to compile the classes, build the assembly, and (optionally) create the XML file that can be used to generate API documentation. See [“Building the Assembly” on page 16-20](#).
5. (optional) Complete the API documentation using a third-party utility. See [“Generating API Documentation” on page 16-27](#).

The genxml Utility

The **genxml** utility creates a Synergy XML file from SMC method definitions and repository structure definitions. This is an intermediate step in creating the C# classes.

This utility runs on all supported Synergy/DE platforms. **Genxml** is installed in the DBLDIR directory.



The **genxml** utility checks structure sizes in the SMC against the corresponding structures in the repository and reports a warning if there are discrepancies. Although the XML file is generated anyway, you should use the MDU's Verify Catalog utility to update the structure sizes in the SMC. (See [“Verifying Repository Structure Sizes and Enumerations” on page 2-42](#).) Failure to do so can cause errors at runtime (because the structure information in the component will differ from that in the SMC).

Creating Synergy .NET Assemblies

Creating an Assembly from the Command Line

Syntax `dbr genxml -f xmlFilename -i intName [-a altIntName]
[-d targetDir] [-s smcDir] [-m rpsMain -t rpsText] [-n]
[-v msgLevel] [-?]`

Arguments -f *xmlFilename*

The name to use for the XML file. This name will also be used for the assembly. You can include the complete path if desired. The extension “.xml” will be appended to this filename if you don’t specify an extension.

-i *intName*

Name of the interface from the SMC to include in the XML file. You may pass multiple interface names; each must be preceded with the -i option. A C# class will be created for each interface specified. Remember, the interface name is case sensitive.

-a *altIntName*

(optional) Alternate interface name. Use this name for the interface previously specified with the -i option. **Genxml** uses the associated -i interface to pull methods from the SMC; the alternate name is included in the XML file and is used as the class name when **gens** is run. If you pass multiple interface names, each may have an alternate name. Each alternate name must be preceded with the -a option. See the examples on [page 16-15](#).



When using alternate names, sequence matters. The -a option must follow the -i option that it applies to. You may specify multiple interface names, and each may have an associated alternate name immediately following it.

-d *targetDir*

(optional) The target directory for the XML file. If not passed, the XML file is created in the directory specified in the -f option. If no directory is specified with -f, the file is created in the current directory.

-s *smcDir*

(optional) Directory where the SMC files (**cdt.is?** and **cmpdt.is?**) are located. If not passed, DBLDIR is used.

-m *rpsMain*

(optional) Full path to the repository main file that contains the structures referenced in the SMC. Use with -t. This option is used only if you are passing structures as parameters. If not passed, **genxml** uses the environment variable

RPSMFIL to determine the name of the repository main file; if that is not set, it uses **RPSDAT:rpsmain.ism**. If RPSDAT isn't set, **genxml** looks in the current directory for **rpsmain.ism**.

-t *rpsText*

(optional) Full path to the repository text file that contains the structures referenced in the SMC. Use with **-m**. This option is used only if you are passing structures as parameters. If not passed, **genxml** uses the environment variable RPSTFIL to determine the name of the repository text file; if that is not set, it uses **RPSDAT:rpstext.ism**. If RPSDAT isn't set, **genxml** looks in the current directory for **rpstext.ism**.

-n

(optional) Indicates that you want to use the value in the Repository Alternate name field instead of the value in the Name field as the property or field name. This option pertains only if you are passing structures as parameters. If not passed, the field name in the structure becomes the property/field name in the C# class. If passed, the value in the Alternate name field is used if it exists; else, the value in the Name field is used.

-v *msgLevel*

(optional) Level of verbosity in messages:

0 = no messages

1 = error messages and warnings

2 = everything included in level 1, plus success messages (default)

3 = everything included in level 2, plus return codes and the location of the SMC and repository files

-?

(optional) Displays a list of options and the version number for **genxml**.

Examples This example creates a Synergy XML file named **ConsultIt.xml**. This will also be the name of the assembly. The XML file will include information about two interfaces, AppLogin and Consultant. The target directory for the XML file is c:\work, which is also where the SMC files are located.

```
dbr DBLDIR:genxml -f ConsultIt -i AppLogin -i Consultant  
-d c:\work -s c:\work
```

Creating Synergy .NET Assemblies

Creating an Assembly from the Command Line

This example uses alternate interface names. Instead of classes named AppLogin and Consultant, the assembly will have classes named Login and Consult.

```
dbr DBLDIR:genxml -f ConsultIt -i AppLogin -a Login  
-i Consultant -a Consult -d c:\work -s c:\work
```

On OpenVMS, you'll need to define **genxml** as a foreign command and then execute it. In the example below, we quoted the XML filename and the interface names to preserve the case.

```
$ GENXML:==$DBLDIR:GENXML  
$ GENXML -F "ConsultIt" -I "AppLogin" -I "Consultant" -  
-D SYS$WORK: -S SYS$WORK:
```

The gens Utility

The **gens** utility takes as input the Synergy XML file created by **genxml** and outputs the following:

- ▶ C# source files. By default, there will be one C# class file for each interface you specified when running **genxml**, as well as a C# class file for each structure and group in the interface. There will also be a C# interface file generated for each interface you specified when running **genxml**.
- ▶ A batch file named *xmlFilename.bat*, which is used to compile the classes, generate the assembly, and create the XML file for the API documentation.
- ▶ A response file named *xmlFilename.rsp*, which is used by the batch file to compile the classes.
- ▶ A file named **AssemblyInfo.cs**, which contains information about the assembly. You can customize this file; see [“Editing Information in AssemblyInfo.cs” on page 16-26](#). Once created, this file will not be overwritten if you regenerate classes.

The **gens** utility is installed in the xfnlNet directory.



The **gens** utility must be installed on your local machine. Attempting to run it on a remote machine using a mapped drive will result in a .NET security error.

Syntax `gens -f xmlFilename [-d targetDir] [-n namespace]
[-o outputDir] [-s keyFilename] [-t] [-a] [-g] [-nb]
[-i interfaceName:count] [-v msgLevel] [-?]`

Arguments `-f xmlFilename`

The full path and filename of the XML file generated with **genxml**. If you do not specify the file extension, “.xml” will be appended to the end of the filename.

`-d targetDir`

(optional) The directory for the generated files. If not passed, defaults to the My Documents directory. The **gens** utility creates a subdirectory, named with the XML filename, within *targetDir* and places all the generated files in it.

`-n namespace`

(optional) The namespace that the assembly will use. If specified, all classes generated for the assembly will use this namespace. If not specified, the namespace will be *xmlFilenameNS*.

The namespace is used to ensure that each class is unique. Microsoft recommends that namespaces use the format *CompanyName.ProductName* (e.g., ABCComputers.ConsultPro). The namespace is appended to the beginning of the class name (e.g., ABCComputers.ConsultPro.MyClass).

`-o outputDir`

(optional) The directory that the assembly will be created in. If not passed, defaults to the **-d** directory or, if **-d** was not passed, to the *xmlFilename* subdirectory of the My Documents directory.

`-s keyFilename`

(optional) Full path and filename of the strong name key file that will be used to strong name the assembly. You must create the file using Microsoft’s **sn.exe** utility and place it in the desired location before running **gens**. The **gens** utility will verify that the file exists and then write the path to the batch file. See [“Using Your Own Key File” on page 16-25](#) for more information.

If not passed, *xyfNetLink* .NET will generate public and private keys in a strong name key file named with the assembly name.

`-t`

(optional) Assembly will be delay signed. Use with **-s**. See [“Using Your Own Key File” on page 16-25](#) for more information.

Creating Synergy .NET Assemblies

Creating an Assembly from the Command Line

- a
(optional) Generate an XML file that can be used to generate API documentation. This option adds a command to create the XML file to the batch file. In order to have useful documentation, you must add comments for your methods and parameters. See [“Generating API Documentation” on page 16-27](#) for more information.
- g
(optional) Generate structure members as public fields. If not passed, structure members are generated as properties with private fields. If you are planning to use .NET controls, generate properties; the properties have get and set methods, which can be assigned to .NET controls in Visual Studio. You must also generate properties to take advantage of Repository’s read-only flag and the class Changed property.

If you are not planning to use .NET controls, and don’t need the read-only flag or Changed property, you can generate either fields or properties; using fields may improve performance.
- nb
(optional) Convert binary fields in repository structures to strings. By default, a binary field in a repository structure is converted to a byte array of the size specified in the Repository field definition. This feature was added in version 9.3, and can be used, for example, to store an RFA that can then be used to ensure you update the correct record when data is returned to the server. To retain the pre-9.3 behavior and convert binary fields to strings, use the **-nb** option. This option is not available when creating an assembly from Workbench. See [Appendix B](#) for more information on data type conversion. Note that you should use the procedure described in [“Passing Binary Data” on page 17-22](#), rather than a binary structure field, to pass most binary data.
- i *interfaceName:count*
(optional) Generate multiple classes of the specified interface name. Specify the *total* number of classes desired with *count*. The first instance of the class will be named as usual with the interface name (or alternate name, if specified). Subsequent instances will be named with the interface name followed by a number, which will increment. (See the examples on [page 16-19](#).) You can pass multiple interface names; each must be preceded with the -i option. If -i is not specified for an interface, a single class will be generated. (See [“Using Multiple Copies of the Same Class” on page 17-12](#) for more information on using this feature.)

-v *msgLevel*
(optional) Level of verbosity in messages.
0 = no messages
1 = error messages
2 = error messages and success messages (default)

-?
(optional) Displays a list of options for the utility.

Examples This example creates **.cs** files from **ConsultIt.xml**, and indicates that you want to create an XML file for API documentation. The namespace uses the standard *CompanyName.ProductName* format (in our example, ABCComputers.ConsultPro). A subdirectory named ConsultIt will be created within c:\work, and the **.cs**, **.bat**, and other generated files will be placed there. This example also includes a strong name key file and the **-t** option to delay sign the assembly. This is a continuation of the **genxml** example, so the new files will be named **AppLogin.cs** and **Consultant.cs**.

```
gens -f c:\work\ConsultIt.xml -d c:\work -a  
-n ABCComputers.ConsultPro -s c:\corp\ABCComputersPublic.snk -t
```

The XML file **ConsultIt.xml** contains two interfaces, AppLogin and Consultant. In the following example we use the **-i** option to specify that we want to create multiple classes (three total) from the interface AppLogin. These classes will be named **AppLogin.cs**, **AppLogin1.cs**, and **AppLogin2.cs**. Because we have not specified the Consultant interface with the **-i** option, only one Consultant class will be generated.

```
gens -f c:\work\ConsultIt.xml -d c:\work -a  
-n ABCComputers.ConsultPro -s c:\corp\ABCComputersPublic.snk -t  
-i AppLogin:3
```



Gens also appends numbers to the ends of interface names when it encounters names that differ only in case or a structure name that is the same as an interface name. Consequently, the numbers appended to the additional class files created with the **-i** option may not start with 1. See the note on [page 2-25](#) for more information.

Building the Assembly

Before building the assembly, you may need to edit the generated C# class files or the **AssemblyInfo.cs** file. See [“Understanding the Generated Classes”](#) on page 16-22 and [“Editing the Generated Files”](#) on page 16-26.

The **gens** utility creates a batch file (*xmlFilename.bat*) and places it in the subdirectory created in the directory specified with the **gens -d** option. The batch file includes commands to compile the C# classes, build the assembly, and—if you included the **gens -a** option—generate an XML file that can be used for API documentation.

If desired, you can pass C# compiler commands to **gens** by setting the **SYNCSCOPT** environment variable to any valid C# compiler command before running the batch file. See [SYNCSCOPT](#) in the “Environment Variables” chapter of the *Environment Variables and System Options* manual for more information.

1. Run **vsvars32.bat** to set the environment variables used by .NET. This batch file is part of Visual Studio and should be run whenever you run commands from the command line. If desired, you can run **sdvars.bat** instead. It is installed with the .NET Framework SDK.



To build an assembly on a 64-bit platform (or build on a 32-bit platform to run on a 64-bit platform), set up the environment by running **vcvarsall.bat** with the relevant parameter, such as **amd64**. If desired, you can always run **vcvarsall.bat** instead of **vsvars32.bat**. (Use the parameter **x86** to build on 32-bit.) See your Visual Studio documentation for more information on these batch files.

2. (optional) Set **SYNCSCOPT**.
3. Run the batch file created by **gens**. For example, to continue with our previous scenario:

```
c:\work\ConsultIt\ConsultIt.bat
```

To enable the classes in your assembly to be pooled, you must run the batch file with the **-p** option. For example:

```
c:\work\ConsultIt\ConsultIt.bat -p
```



Using the **-p** option causes the generated procedural classes to be derived from the `ServicedComponent` class. It also changes the status of some methods from public to private or protected, and causes the `getConnect()` and `shareConnect()` methods to be excluded from the procedural classes. Thus, you should use **-p** only when you are certain the object will be pooled. See [“Understanding .NET Pooling” on page 17-26](#) and [“Implementing Pooling” on page 17-28](#) for more information on pooling.

The assembly (named `ConsultIt.dll` in our example) will be placed in the directory specified with the **gens -o** option, if specified, or in the same directory as the generated classes. If you opted to create an XML file to be used for API documentation, the XML file will be placed in the subdirectory under the directory specified with the **gens -d** option.

Understanding the Generated Classes

When you generate C# classes, a .cs file is created for each interface selected from the SMC, as well as for each structure and group within the selected interfaces. These generated classes use the *x/NetLink* .NET classes internally to connect with *x/ServerPlus* and pass data.



See [Appendix B](#) for details on how data types for parameters, return values, and structure fields are mapped from Synergy to .NET.

Procedural Classes

The procedural classes (which are derived from the interfaces in the SMC) contain your own user-defined methods as well as a number of utility methods. You can browse a procedural class using the Object Browser in Visual Studio to see the method signatures.

The following public utility methods are included in every procedural class:

```
connect()  
disconnect()  
getConnection()  
shareConnect()  
debugInit()  
debugStart()  
setUserString()  
getUserString()  
setCallTimeout()
```

The utility methods enable you to establish a connection with *x/ServerPlus* and perform other utility functions, such as setting a call time-out value and running a debug session. See [“Using Your Synergy .NET Assembly” on page 17-9](#) and the method reference on [page 17-38](#) for more information. Note that although they are included in the C# classes, `getConnection()` and `shareConnect()` are not included in the assembly when it is built for pooling. This is because these two methods cannot be used with pooling.

For each procedural class, **gens** creates an associated C# interface. The C# interface is named with the class name, with the letter “I” appended to the beginning of the filename. For example, if the class is named `AppLogin`, the associated C# interface will be named `IAppLogin` and will be in the file `IAppLogin.cs`. See [“Using Multiple Copies of the Same Class” on page 17-12](#) for information on using the C# interfaces.

Structure Classes

The structure classes are built from your structure definitions in the repository. Fields in the repository structure become either properties of the structure class or public fields (if you ran **gens** with the **-g** option or selected “Fields” for “Generate structure members as” in Workbench). By default, the properties or fields are named with the repository field name; see [“Passing Structures as Parameters” on page 1-8](#) for details on overriding the default name.

If there are groups within your repository structures (or fields declared in the repository as “struct” data type), a class will be generated for each group, and each field in the group will be a property/field of that class.

The following public utility methods are included in every structure class:

```
Clone()  
Equals()
```

For more information about the structure and group classes, see [“Using Structures” on page 17-14](#). For more information about the utility methods, see the method reference on [page 17-38](#).

DataTable Classes

If you indicated in the SMC that a structure collection or an ArrayList of structures should be created as a DataTable, in addition to the structure class, your assembly will include a DataTable class, which is named with the structure name plus “DT” (e.g., MystructDT).

Each row in the table consists of a structure class, and each column represents a field, which is usually named with the repository field name. Groups within the structure are “flattened” and included in the table with their parent structure.

The following public utility methods are included in every DataTable class:

```
AddRow()  
GetRow()  
GetRows()
```

For more information about the DataTable classes, see [“Using Data Tables” on page 17-18](#). For more information about the utility methods, see the method reference on [page 17-38](#).

Enumerations

Enumeration types are built from your enumeration definitions in the repository. There will be a .cs file, named with the enumeration name, for each repository enumeration that is referenced as a parameter or return value in the SMC, as well as for those enumerations that are referenced as fields within a structure that is passed as a parameter.

For more information, see [“Using Enumerations” on page 17-21](#).

Custom Attributes

*xy*NetLink .NET uses a custom attribute class to hold metadata about the Synergy routines, such as the type and size of parameters for a method call. At runtime, *xy*NetLink .NET uses reflection to look at the attributes and get this information. It is very important that you *not* edit the custom attributes. Doing so will cause errors at runtime.

Using Your Own Key File

By default, when you build the assembly, *xfNetLink* .NET generates public and private keys in a *strong name key file* (.snk file) named with the assembly name. These keys are then used to sign the assembly when you build your project in Visual Studio.

If signing is not a concern for your application, you can simply use the defaults (i.e., do nothing in this regard). It is likely, however, that your company will want to develop a policy for signing assemblies. In this case, you will need to create your own key file and specify it when you create the assembly (see the steps below).

Depending on your workflow, you may want to use *delayed signing*. Delayed signing assigns the public key when the assembly is built, but enables you to delay the assignment of the private key until you are ready to distribute the assembly. You can use delayed signing for your Synergy assembly only when you use your own key file.

► To use your own key file

1. Before generating C# classes, create an .snk file using Microsoft's Strong Name tool, **sn.exe**, which is included in the .NET Framework SDK.
2. If you are creating the assembly in Workbench, enter the name of your key file in the "Key file" field in the Component Information dialog box for your project. If you want to delay sign the assembly, also check the "Delay sign assembly" option. See ["Creating a Synergy/DE .NET Component Project" on page 16-6](#) for details.
3. If you are creating the assembly from the command line, when you run **gens**, specify the key file name with the -s option. If you want to delay sign the assembly, also specify the -t option. See ["The gens Utility" on page 16-16](#).

A complete discussion of strong naming and delayed signing is beyond the scope of this manual. For additional information, and for instructions on using **sn.exe** to create a key file, see your .NET documentation.

Editing the Generated Files

You may need to edit the generated C# source files to add methods, such as validation, utility, and initialization methods. If you edit the files, take care not to alter the generated code; doing so will cause errors when you try to run your .NET application. You should place your own methods at the end of the file, after the generated code.

You may also want to edit the metadata in the **AssemblyInfo.cs** file (see below) or add comments to the source files for the API documentation (see [“Generating API Documentation” on page 16-27](#)).



If you regenerate classes for the same interface, the C# source files will be overwritten and any modifications will be lost. The **AssemblyInfo.cs** file is *not* overwritten when classes are regenerated.

Editing Information in AssemblyInfo.cs

The **AssemblyInfo.cs** file contains information about the assembly, which you can customize. This information displays when you view the generated DLL's properties (that is, right-click on the DLL in Windows Explorer and select Properties).

The generated **AssemblyInfo.cs** file includes default values for some of the attributes, as shown in the sample below. If desired, you can add values for the other attributes. For example, you may want to implement a version numbering system. We do not recommend changing the value for **ApplicationActivation**.

```
ApplicationName("ConsultIt")
ApplicationActivation(ActivationOption.Server)
ApplicationAccessControl(false)
AssemblyTitle("")
AssemblyDescription("")
AssemblyConfiguration("")
AssemblyCompany("")
AssemblyProduct("")
AssemblyCopyright("")
AssemblyTrademark("")
AssemblyCulture("")
AssemblyVersion("1.0.0.0")
AssemblyKeyName("")
```

Generating API Documentation

You can create API documentation for the developers who use your Synergy assembly. To do this, you will need to do the following:

1. Add documentation comments for your methods, return values, and parameters.
2. Generate an XML file when you build the assembly.
3. Use a third-party application to create documentation from the XML file.

Adding Documentation Comments

To produce useful documentation, you must provide a description for each of your user-defined methods. (We include documentation comments for the *xyNetLink* .NET utility methods in every class.)

If you are using the MDU to populate the SMC, enter comment text in the description fields in the MDU as you define methods. The MDU includes fields for method description, return value description, and parameter description. See [“Creating New Methods” on page 2-22](#) and [“Defining Parameters” on page 2-28](#) for instructions on entering data in these fields.

If you are attributing your code and using the XML file output by **dbl2xml** to populate the SMC, use the documented syntax to include comment text for methods, return values, and parameters in the Synergy source file. See [“Documentation Comments” on page 2-19](#).

When you generate C# classes, the descriptions in the SMC are included in the generated files as XML documentation comments. If a method does not have a method description in the SMC, a “To Do” comment is inserted in the generated file. If a return value description is not provided in the SMC, no text is inserted in the file for the associated XML tag. If a parameter description is not provided in the SMC, the parameter name is used as the comment text.

For structures passed as parameters, the field description in the repository is included as the XML documentation comment for the property (or field) in the generated class file. If a field does not have a description in the repository, the comment “***Field To Do***” is inserted in the generated file.

It is also possible to manually edit the C# source files to include documentation comments or to add additional formatting tags to create the desired output. Keep in mind, though, that any changes you make to the .cs files will be lost if you regenerate the C# classes. See [“To manually add XML documentation comments”](#) below for instructions.

► To manually add XML documentation comments

1. After generating C# classes, open the class file(s) and find the “To Do” comments. There is a “***To Do*** Add method description” comment before every user-defined method for which there was no description text found in the SMC. There is a “***Field To Do***” comment before every property or field in the structure classes for which there was no description in the repository.
2. Replace the “To Do” lines with descriptions of the methods and fields. Add comments for the <param> and <returns> tags if desired.
3. Save the file(s).

Generating an XML File

Select the “Generate API documentation” option in Workbench or use the **gens** -a option on the command line to indicate that you want to generate an XML file when you build the assembly.

When you build the assembly, a file named *assembly_nameAPI.xml* will be generated and placed in the same directory as the assembly.



When the C# classes are generated, the XML tags necessary to produce documentation (e.g., <summary>) are included in the source files regardless of whether you have indicated that you want to generate API documentation. If you do indicate in Workbench or with the command line option that you want to generate API documentation, a command to create the XML file is added to the batch file.

Creating the API Documentation

A third-party product, such as Sandcastle or NDoc, is required to create the API documentation. These products use the *assembly_name*API.xml file and the *assembly_name*.dll file to create API documentation in various formats, including HTML and compiled HTML (.chm).

- ▶ Sandcastle is a Microsoft product that can be downloaded for free. As currently distributed, it has only a command-line interface, but several developers have written GUIs for it. For more information, including download links, see <http://www.sandcastledocs.com>.
- ▶ NDoc is an open source application that can be downloaded from Sourceforge.net. NDoc 2005, which supports Visual Studio 2005, can be downloaded from <http://sourceforge.net/projects/ndoc05/>. NDoc3, which can be used with Visual Studio 2008, can be downloaded from <http://sourceforge.net/projects/ndoc3/>. Since NDoc is an open source application, you should check the websites to determine the current state of development.

Calling Synergy Routines from .NET

This chapter explains how to deploy your Synergy assembly and the other tools necessary to create a distributed application with .NET and Synergy, how to create an application configuration file, and how to use your assembly in your client application. It also includes instructions for using .NET pooling, as well as a method reference for the utility methods included in your C# classes.

Deploying Your Distributed Application

This section tells you what needs to be installed and configured on the *xyServerPlus* machine and the client machine. You will need to refer to this section when deploying your assembly so that your .NET developer can use it, as well as when creating the installation program for your distributed application.

Deploying the Server

For instructions on deploying the server portion of your distributed application, see [“Deploying the Server” on page 3-44](#).

Deploying the Client (for Development)

Follow these steps to deploy your Synergy assembly and related files for the purpose of developing your .NET application.

1. Install Visual Studio and the .NET Framework SDK.
2. Install *xyNetLink* .NET. During installation, *xfnlnet.dll* will be registered in the GAC (global assembly cache) and installed as a native image.
3. Copy your Synergy .NET assembly to the client machine. Put it in a location where your application can reference it and, if necessary, register it in the GAC. There are two instances in which your assembly must be registered in the GAC:
 - ▶ If you want the assembly to be public.
 - ▶ If you are using pooling *and* passing structures as parameters.

4. (optional) Copy the files for the API documentation to the client machine. These files can go in any convenient location.

At this point, the .NET developer can get to work creating an application using your Synergy assembly. When the application is ready to test, complete the remaining deployment steps.

5. If you are using a configuration file, use the Synergy .NET Configuration utility to create or edit the configuration file for your application. (In some cases, you may need to create the file with Visual Studio; see the note on [page 17-5](#).) See “[Using an Application Configuration File](#)” on [page 17-3](#) for more information about configuration files and instructions on using the utility.
6. If you are using pooling, create, configure, and start the pool. See “[Understanding .NET Pooling](#)” on [page 17-26](#) and “[Implementing Pooling](#)” on [page 17-28](#).

Deploying the Client (at a Customer Site)

Follow these steps when you deploy your completed .NET application at a customer site.

1. Install the .NET Framework runtime, **dotnetredist.exe**, available on the Synergy/DE software CD or on the Microsoft website.
2. Install *x/NetLink* .NET. During installation, **xfnlnet.dll** will be registered in the GAC and installed as a native image.
3. Install your .NET application and all the assemblies it uses. If necessary, register the assemblies in the GAC. There are two instances in which your assembly must be registered in the GAC:
 - ▶ If you want the assembly to be public.
 - ▶ If you are using pooling *and* passing structures as parameters.
4. If you are using a configuration file, use the Synergy .NET Configuration utility to edit the application configuration file for your application. See below for instructions. Note that you can create the config file ahead of time, install it with your application, and then edit it as necessary for each customer site.
5. If you are using pooling, create, configure, and start the pool. See “[Implementing Pooling](#)” on [page 17-28](#).

Using an Application Configuration File

You have the option of using an application configuration file to specify configuration information—such as the host name and port, time-out values, and log file settings—for your .NET client application. Your *xy*NetLink .NET distribution includes the Synergy .NET Configuration utility, which is used to create and edit configuration files.

What is an application configuration file and when should I use one?

An *application configuration file* (config file, for short) contains settings that are used by .NET applications. Default values for the configuration settings used by *xy*NetLink .NET are built into the program. These *program defaults*, shown in the table below, cannot be changed. If they are suitable for your application, you do not need to use a config file. If they are not suitable, you must use a config file to override some or all of the settings.

Configuration setting	Program default
connect timeout	120 seconds
host	localhost
initialize timeout	30 seconds
logfile	c:\xfnlnet.log
logging	none (i.e., logging is off by default)
pool return	false
port	2356
single log file	off

*xy*NetLink .NET employs a standard .NET configuration file, which uses XML to record configuration settings. In general, there is one config file per application. (This rule does not always strictly apply; see [“Naming the configuration file”](#) below.) Within the config file, you can create default settings that will be used by all assemblies that use that particular configuration file. These default config file

settings override the program defaults. In addition, you have the option of creating settings for individual procedural classes within your assemblies. Settings at the class level override the default settings.



If you use an explicit `connect()` to establish the connection with `xfServerPlus`, you can override the settings (specified in the config file or the program defaults) for host and port at runtime. See [step 4 on page 17-10](#) and the method reference on [page 17-38](#) for more information.

Naming the configuration file

`xfNetLink .NET` uses the standard .NET naming conventions for config files. Your file must adhere to these naming conventions in order for `xfNetLink .NET` to use it.

- ▶ If you are using pooling, the file must be named **dllhost.exe.config** and reside in the `winnt\system32` or `windows\system32` directory. This means that if you have multiple applications residing on the same machine, they must all use a single config file. If those multiple applications require different configuration settings, you must implement the settings at the class level.
- ▶ For a web application that does not use pooling, the file must be named **web.config** and reside in the root directory for the web application. If you have multiple web applications residing on the same machine and want each one to use a different config file, you must ensure that the applications reside in different directories.
- ▶ For a Windows (i.e., non-web) application that does not use pooling, the file must be named with the application filename (including extension) plus the **.config** extension. It must reside in the same directory as the application. For example, if your application were named **myApp.exe**, you would name the config file **myApp.exe.config**.

Creating and Editing Configuration Files

The Synergy .NET Configuration utility (**synnetcfg.exe**) enables you to create and edit application configuration files for your Synergy .NET applications. The config files created by the Synergy .NET Configuration utility use XML. Although you may need to edit the config file to add additional information required by your application, we *do not* recommend manually editing the *xfNetLink* .NET portions of the file.



When you create a Visual Studio project for a web application, Visual Studio will create a config file named **web.config** for you. We recommend that you use this file (assuming your web application does not use pooling) because it may contain information that is required by your application. To add the *xfNetLink* .NET information to the file, just open the file in the Synergy .NET Configuration utility and select “Yes” when prompted to add the information.

During development of a Windows (non-web) application that does not use pooling, it may be necessary to create the config file in Visual Studio and add it to the project. You can then use the Synergy .NET Configuration utility to add the *xfNetLink* .NET information to the file.

► To start the utility

From the Windows Start menu select Programs > SynergyDE > Utilities > Synergy .NET Configuration.

You can also run the utility from the menu option in Workbench. Select Synergy/DE > Utilities > Synergy .NET Configuration.

► To open an existing config file

1. Select File > Open or click the Open config file toolbar button.
2. Navigate to the location of the file, select it, and click Open.
3. When you are done editing, select File > Save or click the Save config file toolbar button.

Calling Synergy Routines from .NET

Using an Application Configuration File

► To create a new config file

1. Select File > New or click the New config file toolbar button.
2. Configure settings for the default class or add a class and configure settings for it. See [“To add a class to a configuration file”](#) and [“To configure settings for a class”](#) below for instructions.
3. When you are through editing the settings, select File > Save As or click the Save config file toolbar button. Give the file a valid name. See [“Naming the configuration file” on page 17-4](#).

► To add a class to a configuration file

1. With a config file open, click the Add Class button.
2. Type the name in the Class name field. The class name is case sensitive. Be careful to type the name correctly; the Synergy .NET Configuration utility does not validate the name you enter.
3. Click OK. You can now configure settings for the class; see below for instructions.

► To configure settings for a class

1. Select the class from the list of Synergy Classes. Any settings currently defined for the class will display in the Class Settings list.
2. Click the Add button to display the Add Class Setting dialog box.
3. In the Key field, select the setting you want to add.
4. In the Value field, specify the value for the selected key. Refer to the table on [page 17-7](#).
5. Click OK. Repeat this procedure for each setting you want to add.

► To modify a setting

1. Select the class from the list of Synergy Classes. The current settings for that class will display in the Class Settings list.
2. In the Class Settings list, select the setting you want to change.
3. Click the Modify button.
4. In the Modify Class Setting dialog box, type or select a new value for the key. See the table above for details on key values.
5. Click OK.

Class Settings	
Key	Set value to
connect timeout	The number of seconds you want <i>xfNetLink</i> to wait for an acknowledgment from the session started by the logic server in <i>xfServerPlus</i> . (This is 'B' in figure 5-1 on page 5-4 .) The default for normal operation is 120 seconds; for debug it is 600 seconds. Both the normal time-out and the debug time-out will change to the value you specify here. This value is also used to set the request for session time-out ('A' in figure 5-1 on page 5-4).
host	The name or IP address of your <i>xfServerPlus</i> machine.
initialize timeout	(pooling only) The number of seconds you want <i>xfNetLink</i> to wait for a return from a remote call to <i>xfServerPlus</i> when the pool is being started. This value controls a special call time-out that applies only to the <i>Initialize()</i> method. The default is 30 seconds. After the <i>Initialize()</i> method is called, the call time-out is reset to 30 minutes. See "Setting a Call Time-Out" on page 17-24 for information on the regular call time-out.
logfile	The path and filename of the file you want to use for client-side logging. Do not use logicals to specify the filename location.
logging	The information you want logged. When you select logging in the Key field, the Logging Options become available. Check the boxes to indicate what information you want logged. See "Using Client-Side Logging" on page 18-7 for more information and a sample log. ▶ None. Turns logging off for this class. You can also turn off logging by deleting the logging setting. However, this option enables you to have logging on by default but off for a specific class. ▶ All. Includes all types of logging listed below. ▶ Errors. Logs errors only. ▶ User-defined methods. Logs calls to methods in the SMC. Includes parameters passed in and shows what is in the parameter after the call. ▶ Utility methods. Logs calls to the utility methods in the class (e.g., <i>connect()</i>, <i>disconnect()</i>). Includes parameters passed in and out. ▶ Configuration settings. Logs messages regarding configuration settings. ▶ Packets. Logs the complete packets that are sent and received on the client; includes date/time stamp. If encryption is enabled, the log displays a string of 10 asterisks instead of the packet contents.

Calling Synergy Routines from .NET

Using an Application Configuration File

Class Settings (continued)	
Key	Set value to
maxport	 The port number range feature has been deprecated; these settings have no effect.
minport	
pool return	(pooling only) When you select pool return in the Key field, the Pool Options become available. Select the “Return objects to pool” checkbox if you want the objects instantiated from this class returned to the pool after they are used. See “Reusing Objects” on page 17-27 for more information.
port	The port that xfServerPlus is listening on. The port number must be in the range 1024 through 65534.
single log file	“On” to use a single log file for all sessions. If not set, or if set to “off”, a separate log file is used for each class that instantiates a connection to xfServerPlus, and a date/time stamp is appended to the end of the logfile name (before the file extension) to differentiate the logs. See “Using Client-Side Logging” on page 18-7 for additional information.

► To delete a setting

Deleting a setting from a class causes the class to use the default value for that setting or, if there is no default defined in the config file, to use the program default.

1. Select the class from the list of Synergy Classes.
2. In the Class Settings list, select the setting you want to delete.
3. Click the Delete button, and then click Yes at the confirmation prompt.

► To delete a class

Deleting a class removes the class and all its settings from the config file. As a result, this class will use the settings from the default class or, if there is no default class, it will use the program defaults.

1. Select the class from the list of Synergy Classes.
2. Click the Delete Class button, and then click Yes at the confirmation prompt.

Using Your Synergy .NET Assembly

We have included examples in both C# and—where there is a sufficient difference in the code—in VB.NET. If you are using pooling, also refer to [“Writing Code That Uses Pooled Objects” on page 17-35](#).

1. Reference the Synergy assembly in your Visual Studio project

Before using the assembly, you’ll need to add a reference to it in your Visual Studio project. After you’ve referenced the assembly, you can view its methods using Visual Studio’s Object Browser. You’ll see your own methods, along with the *x/NetLink* .NET utility methods.

If any of your methods pass structures as parameters, you must also add a reference to *xfnlnet.dll* in your project.

If you are using pooling, you must also add a reference to *System.EnterpriseServices* in your project. The *x/NetLink* .NET classes use this namespace.

2. Designate the namespace(s) you are using

In your C# code, include the `using` keyword, followed by the namespace(s) of your assemblies, so that you can more easily access the classes in your assembly. For example:

```
using ABCComputers.ConsultPro;
```

If you are using VB.NET, use the `Imports` keyword to designate the namespace(s) you will be using. For example:

```
Imports ABCComputers.ConsultPro
```

3. Instantiate an instance of the procedural class

Your Synergy .NET classes can be instantiated using the standard mechanism for the language you are using. The examples below show how to instantiate an `AppLogin()` object named “userSess”.

For example, in C#:

```
AppLogin userSess = new AppLogin();
```

For example, in VB.NET:

```
Dim userSess As New AppLogin()
```

4. Connect to *xfServerPlus*

There are several ways to establish a connection with *xfServerPlus*.

- Use `connect()`. This is the recommended method. It enables you to make several calls using the same connection before disconnecting. There are several advantages to using `connect()`: it offers improved performance; you can maintain state between calls; and you can share this type of connection with other objects (see next bullet).

For example, in C#:

```
userSess.connect();
```



See the method reference on [page 17-38](#) for an additional `connect()` method, which enables you to specify the host and port at runtime.

- Share a connection. Two or more objects can share a connection. This method improves performance because several objects are sharing the same *xfServerPlus* session rather than each object making its own connection. To share a connection, you must first establish it with `connect()`, and then use the `getConnect()` and `shareConnect()` methods.

In the examples below, we instantiate two new procedural classes and use the `connect()` method to establish a connection for one of them. We then instantiate the `aConnection` object to hold the connection and use the `getConnect()` method of the `userSess` object to get it. Finally, we call the `shareConnect()` method of the `userRoutines` object and pass the `aConnection` object. You can pass the same `aConnection` object multiple times to share the connection among several objects.

In C#:

```
AppLogin userSess = new AppLogin();  
Consultant userRoutines = new Consultant();  
userSess.connect();  
object aConnection = null;  
aConnection = userSess.getConnect();  
userRoutines.shareConnect(aConnection);
```

In VB.NET:

```
Dim userSess As New AppLogin()  
Dim userRoutines As New Consultant()  
userSess.connect()  
Dim aConnection As Object  
aConnection = userSess.getConnect()  
userRoutines.shareConnect(aConnection)
```

- Create the connection automatically. When you make a call using one of the Synergy methods in your assembly, the connection is created automatically and then disconnected when the call is complete. This is the easiest way to connect because it makes access to *xyServerPlus* completely transparent. However, this method does not allow you to maintain state between calls. In addition, it requires more overhead because a connection is opened and closed for each call. If you use this method and experience performance problems, you may want to use an explicit `connect()` instead.

In the examples below, the `userRoutines` object is instantiated, and then the `postCharge()` method is called, without first calling the `connect()` method.

In C#:

```
Consultant userRoutines = new Consultant();  
userRoutines.postCharge(charge, return_msg);
```

In VB.NET:

```
Dim userRoutines As New Consultant()  
userRoutines.postCharge(charge, return_msg)
```

5. Invoke methods in the component

Make calls to your Synergy methods and pass the necessary parameters. If you generated API documentation for your assembly, it will include the information necessary to use the methods, such as the parameter data types. In addition, Visual Studio's IntelliSense® will show you the method signature as you code.



Parameters that were not flagged as required in the SMC were converted to required parameters when you generated the C# classes because you must always pass all parameters in .NET.

Calling Synergy Routines from .NET

Using Your Synergy .NET Assembly

The examples below show how to invoke the `login()` method in the `userSess` object.

In C#:

```
string userID = "MFranklin";
string password = "abc123";
userSess.login(userID, password);
```

In VB.NET:

```
Dim userID As New String("MFranklin")
Dim password As New String("abc123")
userSess.login(userID, password)
```

6. Disconnect from *xfServerPlus*

If you connected to *xfServerPlus* using the `connect()` method, you must disconnect using the `disconnect()` method. If you have multiple objects sharing a connection, *xfServerPlus* will not completely close the connection and release the license until all objects are disconnected. Although an explicit disconnect is not required when you use an automatic connection, you may want to call `disconnect()` before exiting your application to ensure the connection is correctly disposed of.

For example, in C#:

```
userSess.disconnect();
```

Using Multiple Copies of the Same Class

By specifying a quantity in the Component Information dialog in Workbench, or by using the `gens -i` option, you can create multiple copies of a single class from the method definitions in the SMC. The purpose of this feature is to enable you to have identical objects that can access different *xfServerPlus* servers using settings in an application configuration file. This feature can be used either with or without pooling, though it is probably most useful when used with pooling.

For example, you might want to use a single website for several customers. Each customer's users need to be able to log in on the common web page, but then access separate sets of data. If you run *xfServerPlus* on a separate port for each customer, you can use duplicate objects to direct users from customer A to port 2356, users from customer B to port 2357, and so on. Users would be directed to the correct port at log-in, and your application code would be the same for each customer.

The duplicate classes are named with the interface name (or alternate name, if specified), with a number appended to the end. For example, if the interface is named **AppLogin**, and you specify that you want to create a total of three classes, they will be named **AppLogin**, **AppLogin1**, and **AppLogin2**. The C# interface that is generated for each class (and included in the assembly) enables you to easily access the methods in the duplicated classes.

To create an application that uses multiple copies of the same class, you'll need to do the following in addition to the steps you'd normally perform:

- ▶ If you use Workbench, specify the total number of classes you want to generate for each interface in the Component Information dialog. See the instructions for the Qty field on [page 16-9](#) for details.

If you create the assembly from the command line, use the **gens** -i option to specify the interfaces for which you want to create multiples classes and the number of classes to create. See the **gens** syntax on [page 16-16](#) and the examples on [page 16-19](#) for more information.
- ▶ Set the host name and port number for each copy of the class using the Synergy .NET Configuration utility. See “[Setting options in the Synergy .NET Configuration utility](#)” on [page 17-28](#).
- ▶ When writing your client code, use the C# interface included in the assembly to access the methods in the duplicated classes. There is a code sample that illustrates this in “[Writing Code That Uses Pooled Objects](#)” on [page 17-35](#).
- ▶ If you are using pooling, when you run **regsvcs**, all of the classes will be included as components in the COM+ application that is created. You'll need to set the pooling properties (pool size, etc.) for each component. See “[Implementing Pooling](#)” on [page 17-28](#) for more information.

Using Structures

Repository structures passed as parameters to your Synergy methods are included in your Synergy assembly as classes. There will be a separate class for each structure, named with the structure name. The fields in the repository structure will become either properties of the structure class or public fields (if you ran **gens** with the **-g** option or selected “Fields” for “Generate structure members as” in Workbench).



See [Appendix B](#) for details on how data types for structure fields are mapped from Synergy to .NET.

When you generate structure members as properties, fields that are flagged as read-only in Repository will be generated as read-only properties and will therefore have a “get” method but no “set” method. The read-only flag is not honored when you generate structure members as public fields.

For example, say you have this repository structure, which is passed as a parameter to the `login()` method.

```
user
  fname      ,a25
  lname      ,a25
  maxrate    ,d18.2
  group      address    ,a
    street    ,a20
    city      ,a20
    state     ,a2
    zip       ,d9
  endgroup
```

In your assembly, you’ll see a class named “User” with the properties/fields `Fname`, `Lname`, `Maxrate`, and `Address`. (Note that the first letter of each field name is capitalized.) You’ll also see a class named “Address” in the assembly, which represents the Address group within the User structure. (“Address” is the name of the field in the repository that references this group.) The `Address` class has four properties/fields—`Street`, `City`, `State`, and `Zip`. Under normal circumstances, you

won't need to access the `Address` class directly. When you instantiate a new `User` object, an `Address` object is automatically instantiated. (Fields defined as struct data type in the repository are treated the same as group fields.)



By default, the properties/fields are named with the repository field names. See [“Passing Structures as Parameters” on page 1-8](#) for details on overriding the default name.

To access the properties/fields, instantiate an object for the `User` class and assign values to the properties/fields as shown in the examples below.

In C#:

```
User myData = new User();
myData.Fname = "Mickey";
myData.Lname = "Franklin";
myData.Maxrate = 150.00;
myData.Address.Street = "2330 Gold Meadow Way";
myData.Address.City = "Gold River";
myData.Address.State = "CA";
myData.Address.Zip = 956704471;
```

In VB.NET:

```
Dim myData As New User()
myData.Fname = "Mickey"
myData.Lname = "Franklin"
myData.Maxrate = 150.00
myData.Address.Street = "2330 Gold Meadow Way"
myData.Address.City = "Gold River"
myData.Address.State = "CA"
myData.Address.Zip = 956704471
```

Then, pass the `myData` object when you call the `login()` method.

In C#:

```
int retVal = 0;
retVal = userSess.login(userID, password, myData);
```

In VB.NET:

```
Dim retVal As Integer
retVal = userSess.login(userID, password, myData)
```

For more information on passing structures as parameters, and for details on how overlays are handled, see [“Passing Structures as Parameters” on page 1-8](#).

Using the Clone() and Equals() methods

Your generated structure class includes the utility methods `Clone()` and `Equals()`. The `Clone()` method creates an exact duplicate (including data) of the class on which it is called. The `Equals()` method tests whether the data in two structure classes is the same.

In the example below we show how to clone a structure class and then compare it to the original.

```
//Instantiate an instance of Mystruct class
Mystruct Client = new Mystruct();

//Fill data fields of the class
Client.Fname = "Fred";
Client.Lname = "Friendly";
Client.ID = "FF1234";

//Call Clone method on Client class to create duplicate.
Mystruct Client2 = Client.Clone();

.
. // Do some processing here
.
//Call Equals method on cloned class, passing the original
// class.
bool isEqual = Client2.Equals(Client);
```

See also the method reference on [page 17-38](#).

Using the Changed property

You can use the `Changed` property to determine whether data in a structure class has changed since the structure class was created or was returned to the client from the server. The `Changed` property is included in structure classes when structure members are generated as properties (rather than fields).

The `Changed` property returns a Boolean value of true if the value of any property's data field in the structure class has been changed by calling the set method. If no values have changed, it returns false. If you need to test whether a specific field has changed, use either the `Clone()` and `Equals()` methods or the `Original` property.

For example,

```
//Instantiate an instance of Mystruct class
Mystruct Client = new Mystruct();

//Check for changes in the structure's data
bool isChanged = Client.Changed();
if (isChanged);
.
. //Do something here if returns true
.
```

Using the Original property

You can use the Original property to store a copy of a structure within the object. The Original property is included in all structure classes; it is null when created and remains so unless you use it.

The Original property serves a function similar to that of the Clone() method, in that it enables you to compare a copy of a structure with the original to see what (if any) fields have changed. (By contrast, the Changed property simply tells you that something in the structure class has changed.) The difference is that using the Original property stores the copy within the object, whereas with the Clone() method, the copy is separate. This means Original is useful if you need to pass the object to another method while maintaining the copy of the structure. If you continue to use the object after the original structure has served its purpose, you may want to set Original back to null to free up memory.

In the example below, we save a copy of the Client structure class, then check later to see if a field has changed.

```
//Instantiate an instance of Mystruct class
Mystruct Client = new Mystruct();

//Fill data fields of the class
Client.Fname = "Fred";
Client.Lname = "Friendly";
Client.ID = "FF1234";

//Use Clone method to populate Original property.
Client.Original = Client.Clone();
.
. // Do some processing here
.
//Get original structure
```

Calling Synergy Routines from .NET

Using Your Synergy .NET Assembly

```
MyStruct ClientOrig = Client.Original;

//Check to see if ID has changed
If ClientOrig.ID == Client.ID;
.
. // Do something here if changed
.
```

Using DataTables

If your assembly includes a structure collection or an `ArrayList` of structures created as a `DataTable`, in addition to the structure class, you will have a class named with the structure name plus “DT”. For example, if the structure is named `Struct1`, your assembly will have a structure class named `Struct1` and a `DataTable` class named `Struct1DT`. The `TableName` property of the `DataTable` class (`DataTable.TableName`) uses the structure name.

The generated Synergy `DataTable` extends the .NET Framework class `System.Data.DataTable`; consequently, you can use the methods available in that class to retrieve and manipulate data in the table. You can also use the utility methods we supply to manipulate the rows within the `DataTable` as structure classes. (See “[Methods](#)” and “[Examples](#)” on page 17-20.)

When a `DataTable` is associated with a structure collection, it must be an “Out” parameter, passing data from Synergy to the client. A `DataTable` associated with an `ArrayList` parameter can be an “In” parameter or an “Out” parameter, but not an “In/Out” parameter.



We do not recommend using `DataTables` for extremely complex structures because of the overhead required to flatten out the structure, send it across the wire, and rebuild it on the other end. An example of an extremely complex structure would be a large structure that contains multiple groups, which in turn contain multiple groups, some of which contain arrayed fields. A large number of duplicate names also contributes to a structure’s complexity.

Column names

Each row in the DataTable consists of a structure class. Each column in the table is a field in that structure, and is named with the repository field name or the alternate field name if “use alternate field names” was selected. (This is the DataColumn.ColumnName property; you should *not* change this property.)

If the repository structure contains a field that is a group or struct data type, it will be included in the same table as its parent structure. The group name and the field name (or alternate field name) are combined to make the column name. For example, if the group field is called Activatedate and has three fields named Month, Day, and Year, the columns would be named thus:

```
Activatedatemonth  
Activatedateday  
Activatedateyear
```

If a group or struct field contains a subgroup, both the group name and the subgroup name, along with the field name, are concatenated to form the column name. For example, if the above Activatedate group were a subgroup of the Account group, the columns would be named thus:

```
Accountactivatedatemonth  
Accountactivatedateday  
Accountactivatedateyear
```

If the structure contains an array field, the columns will be named with the field name plus an index position. For example, a three-element array named Phone would result in the following columns:

```
Phone1  
Phone2  
Phone3
```

It is possible that duplicate column names could occur, especially if alternate field names are used. If this happens, a number is appended to the end of the duplicate name to make it unique.

Column caption

In addition to a name, each column also has a caption. (This is the DataColumn.Caption property.) The caption can be used to provide a more descriptive name for the column. If there is a value in the Report hdg (heading) field on the Display tab in Repository, it is used for the caption. If there is no report heading value, the column name is used for the caption. If desired, you can change the Caption property from within your client application.

Primary key column

The primary key for the table (the `DataTable.PrimaryKey` property) is set to the first key in the repository structure that consists of unique data fields. The `PrimaryKey` property is an array of `DataColumn` objects, and may therefore consist of more than one column. If the structure does not have an eligible key, the `PrimaryKey` property is not set. If desired, you can set or change the Primary key property from within your client application.

Methods

Your `DataTable` includes several utility methods that can be used to extract and manipulate rows in the table as structure classes.

- ▶ `AddRow()`, to insert a row at a specific location or add a row to the end of the table
- ▶ `GetRow()`, to return a single row by position
- ▶ `GetRows()`, to return an `ArrayList` of all rows

See the method reference on [page 17-38](#) for additional information about the `DataTable` utility methods. You can also use the methods that are part of the `System.Data.DataTable` class; refer to your .NET documentation. Note that if you cast the Synergy `DataTable` or assign it to a .NET `DataTable`, you will lose access to the utility methods mentioned above.



Adding or removing `DataTable` columns can cause unexpected results if you then try to treat the rows as structure classes. For example, if you add a column to the table and then call the `GetRow()` method, the additional column will simply be ignored. If you remove a column and then call any of the utility methods, an error will occur.

Examples

```
//instantiate an instance of the MyStructure structure. This is
// the structure the DT is associated with.
MyStructure Customer = New MyStructure();

//instantiate an instance of the MyStructure DataTable.
MyStructuredT CustomerDT = New MyStructuredT();

//instantiate an instance of component so that we can call its
// methods.
xfTest myComp = New xfTest();
```

```
//call method and pass the DataTable to be filled. myFillMethod
// has one out parameter.
myComp.myFillMethod(ref CustomerDT);

//GetRow example
//returns row 1 from the CustomerDT DataTable to the Customer
// structure.
CustomerDT.GetRow(ref Customer, 1);

//AddRow example
//instantiate an instance of the MyStructure structure
MyStructure NewCust = New MyStructure;

//Put data in the structure
NewCust.Fname = "Fred";
NewCust.Lname = "Friendly";
NewCust.ID = "FF1234";

//call AddRow method, passing the structure and specifying a row
// position for the new row. New structure is added to existing
// Customer DataTable.
CustomerDT.AddRow(NewCust, 8);

// GetRows example
// create an arraylist
ArrayList myAL = new ArrayList();

//call GetRows method on the existing Customer DataTable and
// fill the arraylist.
myAL = CustomerDT.GetRows();
```

Using Enumerations

Repository enumerations passed as parameters or return values or referenced as a field in a structure passed as a parameter are included in your Synergy assembly as enumeration types, or enums. There will be a .cs file, named with the enumeration name, for each enumeration.

The enumerators (elements) are assigned based on the enumeration members you defined in the repository. The underlying type of each element in the enum is **int**.

If you assigned numerical values to the members in repository, they are used; else, values are assigned automatically starting with 0 and incrementing by 1. When you create a new instance of an enum, it has a default value of the enumerator that

Calling Synergy Routines from .NET

Using Your Synergy .NET Assembly

has been assigned 0, if you do not explicitly assign a value. Consequently, when defining your enumeration in the repository, you should select the most logical default value to be the first member.

The following examples use an enumeration named `Color`, which has members `green`, `blue`, etc. In the first example, the enum is passed as a parameter of the `EnumTest1()` method. In the second example, the enum is a field within the `AcmeCustomer` structure class.

```
// Method call example with an enum parameter
AcmeCompanyComponent acme = new AcmeCompanyComponent();
    acme.Color companyHat = acme.Color.GREEN;
    acme.connect(host, port);
    acme.EnumTest1(ref companyHat);
    acme.disconnect();

// Enum field within structure example
AcmeCompanyComponent acme = new AcmeCompanyComponent();
AcmeCustomer customer = new AcmeCustomer(); // structure
Customer.ColorChoice = acme.Color.BLUE;    // enum field of
                                           acme.Color

acme.connect(host, port);
acme.GetCustomer1(ref customer);
acme.disconnect();
```

Passing Binary Data

You can pass binary data, such as JPEG files, by using a byte array.



Binary fields in structures are converted to byte arrays by default. However, if you want to pass binary data such as JPEG files, you should use the procedure described in this section rather than a binary field in a structure, because the latter requires that you specify a size. See also the description of the **gens -nb** option on [page 16-18](#).

In the MDU, the parameter must be defined as a “Binary (handle)” data type. (If you attribute your code, see [example G on page 2-18](#) for instructions on defining a binary handle.) Your Synergy server routine must declare the argument that receives the data as a memory handle (**i4**; do not use **int**). *x/ServerPlus* will place the data in a memory area and pass the memory handle allocated to that area to your Synergy server routine. (You *must* use the memory handle provided by *x/ServerPlus*; do not attempt to allocate your own.) After the data has been returned to *x/NetLink*, *x/ServerPlus* will free the memory area.

In your client code, if the parameter is defined as “in” only in the SMC, you will need to create a byte array and fill it with data, and then make the method call.

For example, in C#:

```
SynAssembly MyInstance = new SynAssembly();
MyInstance.connect();

byte[] baIn = new byte[67000]; // Create the byte[]
.                               // Fill the byte[]
.
.
MyInstance.BinaryArrayMethod(baIn); // Make method call

MyInstance.disconnect();
```

For “out” only parameters, in your client code you will need to create an empty byte array, and then call the method using the `ref` keyword so that data can be returned. For example, in C#:

```
SynAssembly MyInstance = new SynAssembly();
MyInstance.connect();

byte[] baOut = new byte[0]; // Create empty byte[]
// Make method call using the ref keyword
MyInstance.BinaryArrayMethod(ref baOut);

MyInstance.disconnect();
```

For “in/out” parameters, in your client code you will need to create a byte array and fill it with data, and then call the method using the `ref` keyword so that data can be returned.

For example, in C#:

```
SynAssembly MyInstance = new SynAssembly();
MyInstance.connect();

byte[] baInOut = new byte[67000]; // Create the byte[]
.                               // Fill the byte[]
.
.
// Make method call using the ref keyword
MyInstance.BinaryArrayMethod(ref baInOut);

MyInstance.disconnect();
```

Setting a Call Time-Out



You must explicitly create a connection with the `connect()` method to use `setCallTimeout()`.

Use the `setCallTimeout()` method to set the call time-out value in seconds. The call time-out measures the length of time that the .NET client waits for a return from a remote call to *xfServerPlus* (this is ‘C’ in [figure 5-1 on page 5-4](#)). This time-out is measured for *each* send–receive request between the client and *xfServerPlus*. The default value is 1800 seconds (30 minutes).

Once this value has been set, it will continue to be used for all subsequent calls in the current session until it is reset with another invocation of this method. For example, to set the call time-out to 20 minutes use

```
userSess.setCallTimeout(1200);
```

You can also set a call time-out value for *xfServerPlus*, see [SET_XFPL_TIMEOUT on page 1-31](#).



When you use pooling, there is a special call time-out that applies to the `Initialize()` method only. See [page 17-7](#).

Writing to the *xfServerPlus* Log



You must explicitly create a connection with the `connect()` method to use `setUserString()` or `getUserString()`.

Use the `setUserString()` method to pass a user string that is written in the *xfServerPlus* log. This string is stored, and the value set can be retrieved with `getUserString()`. You can, for example, use this method to write the current client to the log.

To use this method, *xfServerPlus* logging must be turned on (see [“Using Server-Side Logging” on page 3-29](#)) and there must be an entry for the subroutine XFPL_LOG in the SMC. (By default, XFPL_LOG is included in the SMC; see [page 1-33](#) for details.)

For example, if your logon routine stored the user name in a string called “username,” you could set it in the log like this:

```
userSess.setUserString(username);
```

You can then retrieve the string you set. Note that `getUserString()` only gets the string from `setUserString()`; it does not retrieve it from the *x/ServerPlus* log.

For example, in C#:

```
string currUser = "";  
currUser = userSess.getUserString();
```

For example, in VB.NET:

```
Dim currUser As String  
currUser = userSess.getUserString()
```

Understanding .NET Pooling

Pooling enables you to create a “pool” of objects that are active and ready to be used when a client sends a request. You can specify the minimum and maximum pool size, time-out values, and whether objects in the pool should be reused. Depending on the requirements of your application, pooling can significantly improve performance by reducing the time necessary to create objects, establish connections, and perform initialization processing.

Upon starting up, the pool is populated up to the minimum level that you specify. When a client requests an object, the request is satisfied from the objects available in the pool. If no object is available, a new one is created, up to the maximum size of the pool. Once the maximum is reached, requests are queued for a specified length of time.

When the client releases an object, you can specify that it be either returned to the pool or discarded. If it is discarded, causing the number of objects in the pool to drop below the minimum size, a new object will be created.

Object Pooling

The type of pooling supported by *xy*NetLink .NET is referred to as *object pooling*. Object pooling creates a pool of Synergy .NET objects. (These are objects instantiated from the procedural classes in your Synergy assembly; objects instantiated from structure classes are not pooled.) When the pool is created, the objects are instantiated and connected to *xy*ServerPlus. If desired, you can write an initialization method that will be run automatically after the connection is established. In addition, there are several other methods that can be used with pooled objects to perform initialization and cleanup tasks. (See [“Using the Pooling Support Methods”](#) on page 17-32.)

Object pooling is most beneficial when the object is used frequently for a short time and a significant portion of that time is spent acquiring the connection and doing initialization. However, any time there is initialization code that needs to be run, object pooling should improve the performance of your application.

Reusing Objects

After an object has been used, it can either be returned to the pool for reuse or discarded. By default, all objects are discarded after use. Discarding an object releases resources and ensures that the next client request receives a “clean” object. In general, stateless objects may be returned to the pool, while objects with state (that is, those that persist data) should be discarded after use.

To specify that an object be returned to the pool, in the Synergy .NET Configuration utility, select the pool return key and select the “Return objects to pool” checkbox (see [page 17-8](#)). You can also determine at runtime if an object should be reused using the `CanBePooled()` method (see [page 17-34](#)).

Tips for Creating Poolable Objects

To enable the classes in an assembly to be pooled, select the “Support object pooling” checkbox in the Component Information dialog box in Workbench. (Or, if you are working from the command line, specify the `-p` option when you run the batch file to build the assembly.) When you build an assembly with pooling enabled, *all* procedural classes within the assembly will become components in the pool. (Note that if you created multiple instances of the same class, each instance will become a component in the pool.) Depending on your application, you may not want to pool all classes in an assembly. When deciding how to group interfaces into assemblies, you should take into account which classes you want pooled.

Each pooled object represents a connection, which means that it requires an *x/ServerPlus* license. Consequently, when deciding which classes to pool and how large to make the pool, you should ensure that the maximum number of all pooled objects does not exceed the available licenses.

To obtain the greatest benefit from object pooling, when writing your Synergy routines you should separate out the initialization and resource acquisition code that is performed for all clients as a prerequisite to actually doing the work of the object. This code can then go in the `Initialize()` method, to be executed when the object is created. Or, depending on the needs of your application, this code may go in the `Activate()` method to be executed when the object is retrieved from the pool by a client.

Implementing Pooling

This section explains how to set up your client machine for object pooling and how to use the pooling support methods.

Implementation Overview

You must create your assembly and develop your application right from the start with pooling in mind. You'll need to do the following:

1. (optional, but recommended) Write pooling support methods (activate, initialize, etc.) and add them to the SMC. See [“Using the Pooling Support Methods” on page 17-32](#).
2. Create your assembly. If you wrote pooling support methods, include the interface that contains them in the assembly. Select the “Support object pooling” checkbox in the Component Information dialog box in Workbench. (Or, if you are working from the command line, specify the `-p` option when you run the batch file.) See [page 16-11](#) for building from Workbench or [page 16-20](#) for building from the command line.
3. Write your client-side code. See [“Writing Code That Uses Pooled Objects” on page 17-35](#).
4. (optional) Set the desired options in the Synergy .NET Configuration utility. See [“Creating and Editing Configuration Files” on page 17-5](#).
5. Run `regsvcs.exe` to create a pool. See [“Creating a pool” on page 17-29](#).
6. Configure the pool in Component Services. See [“Configuring the pool” on page 17-29](#).
7. Start the pool, and then run your application. See [“Starting the pool” on page 17-31](#).

Setting Up the Client Machine

To set up the client machine, you may want to set some options in the Synergy .NET Configuration utility. Then, you'll create, configure, and start the pool.

Setting options in the Synergy .NET Configuration utility

There are two configuration settings specific to pooling—initialize time-out and pool return. You may want to override the default values for these settings. (See [“What is an application configuration file and when should I use one?” on](#)

page 17-3.) To do so, run the Synergy .NET Configuration utility and specify the necessary settings. You can create settings at the default class level or for individual classes. See “[Creating and Editing Configuration Files](#)” on page 17-5 for more information about specifying settings.

If you need to change any configuration settings after your pool is started, see “[Changing the Pool Configuration](#)” on page 17-31.

Creating a pool

The pool may be created automatically when you build your project in Visual Studio. If it is not, run the **regsvcs.exe** utility from a command prompt to create the pool. You will also need to use this utility to set up the pool when you deploy your completed application at a customer site. The **regsvcs** utility is distributed with the .NET Framework. For example:

```
regsvcs assembly.dll
```

where *assembly* is the name of your Synergy assembly. You must run this command for each Synergy assembly that uses pooling. This command creates a COM+ application (named with the assembly name) and adds the procedural classes in your assembly to the application.

Configuring the pool

1. From Administrative Tools, select Component Services. (On Windows Vista, run %windir%\system32\comexp.msc.)
2. In the tree in the left pane, expand the Component Services node and its subnodes until you see the COM+ Applications node. Expand this node and locate the node named for your Synergy assembly. (See [figure 14-1](#) on page 14-19.)
3. Expand the node for your application, and then expand the Components subnode to display the individual components. There is a component for every class in the assembly. If you created multiple copies of the same class (**gencs -i** option), you will see a component for each copy of the class.
4. Highlight an individual component node (e.g., ConsultIt.AppLogin.1 in [figure 14-1](#)) and select Action > Properties.
5. In the Properties dialog box go to the Activation tab. (See [figure 14-2](#) on page 14-20.)

Calling Synergy Routines from .NET

Implementing Pooling

6. Select the “Enable object pooling” checkbox (it may already be selected), and then enter values for the following:

Minimum pool size. Enter the number of objects you want created at pool start-up. The pool will never drop below this size.

Maximum pool size. Enter the maximum number of objects that you want in the pool at any one time. When deciding how large to make the pool, keep in mind how many *x/ServerPlus* licenses you have available and how many objects you have pooled. The maximum pool size value for all pooled objects should not exceed the number of available *x/ServerPlus* licenses.

Creation timeout. Enter the length of time, in milliseconds, that a request should be queued when all objects in the pool are in use and the pool is at its maximum size. If the request is not satisfied within the specified time, an error is thrown.

7. Select the “Component supports events and statistics” checkbox. This option enables you to view statistics about the pooled object, such as the pool size and the number of objects in use, in the Component Services window.
8. If the “Enable Just In Time Activation” checkbox is selected, clear it.
9. Click OK in the Properties dialog box. Repeat [steps 4](#) through [9](#) to configure each component.



If you have problems with the pool shutting down unexpectedly, try selecting the “Leave running when idle” option on the Advanced tab of the Properties dialog box that is accessed from the COM+ application node (“ConsultPro” in [figure 14-1](#)). Optionally, you may want to increase the “Minutes until idle shutdown” value instead. The pool will wait indefinitely for the first connection, but once a connection to the pool has been made, this timer starts counting.

There are numerous other settings that you can configure for your pool. The recommended configuration will depend upon your specific application. Refer to your COM+ documentation for details on additional settings.

Starting the pool

1. Highlight the node for your COM+ application.
2. Select Action > Start.

If you do not start the pool manually, it will be populated (i.e., started) the first time a client requests an object. Note, however, that this means that the first user will have to wait for the pool to start and the objects to be initialized. Because the primary goal of pooling is to improve performance at start-up, we recommend manually starting the pool.



(Windows 2003/Vista) If you get a permissions error when attempting to start the pool, it may be related to role-based security. To turn off this security check, right-click on the node for the COM+ application you created and select Properties. In the Properties dialog box, go to the Security tab and clear the "Enforce access checks for this application" box. Click OK in the dialog box.

Changing the Pool Configuration

The pool size and creation time-out values are read when the pool is started. If you change these values in the Component Services utility, you must stop and restart the pool for the changes to take effect.

The Synergy .NET Configuration utility settings (host, port, pool return, etc.) are read each time a new object is created and added to the pool. You should restart the pool after changing these settings to ensure that all objects in the pool are using the new settings.

Because restarting the pool will deactivate any objects currently in use, you may want to check the pool status before stopping the pool. To check the status, in Component Services highlight the Components node under your COM+ application node and select View > Status. The number of objects in use displays in the Activated column.

Using the Pooling Support Methods

NetLink .NET enables you to access pooling support methods defined by the COM+ API. These five methods enable you to specify that certain actions be performed at specified times during the object's creation and use. You do not call these methods directly. Rather, they run automatically at various times during the lifetime of the pooled object.

► To use the pooling support methods

1. Write a Synergy routine that performs the desired task. You can name this routine anything you like.
2. Use the Method Definition Utility to add the routine to the SMC. Note the following:
 - In the Method name field, enter the name of the method: Initialize, Activate, Deactivate, CanBePooled, or Cleanup. Because the pooling support methods are called automatically, they must be named correctly.
 - In the Interface name field, specify the interface for the object that this routine applies to.



You can have several routines of the same type if necessary, as long as they are in different interfaces. For example, you might want a separate initialize routine for each of several interfaces. The method name must be as shown below in **"Pooling support methods"**. However, the names of the Synergy routines can vary, so you can include them all in the same ELB if desired.

3. Include the Synergy routine when you build your ELB.
4. Generate classes and build the assembly as you normally would. Select the "Support object pooling" checkbox in the Component Information dialog box in Workbench. (Or, if you are building the assembly from the command line, specify the -p option when you run the batch file.)

Pooling support methods

The methods are listed below in the order in which they are called during the object's lifetime.

Initialize method

`status = Initialize()`

`status`—a `^VAL` value that indicates whether the initialization was a success. Returns 0 for success or 1 for failure.

If the return value is 1, the object is destroyed, an error is logged in the client-side log (if logging is turned on) and the Windows event log, and no other objects in the pool are created. When this happens, the pool will have been started, but there will be no objects in it. To recover from this state, you must stop the pool, fix the problem with the `Initialize()` method, rebuild the ELB, and then restart the pool.

`Initialize()` is called when the object is created. Use this method to prepare the environment by opening files, initializing global data, and so forth. Because `Initialize()` is called when the object is created, this method gets called only once per object, even if the object is reused. Compare with `Activate()`.

Activate method

`Activate()`

`Activate()` is called when the object is requested by a client. This method can be used for code that should be executed when the object is actually used.

Both `Activate()` and `Initialize()` can be used for similar purposes—preparing the environment before using the object. The primary difference between them is that `Activate()` is called *every time* the object is allocated to a client, whereas `Initialize()` is called only *once* when the object is created.

Deactivate method

`Deactivate()`

`Deactivate()` is called when an object is released by the client. It can be used to reset the environment to a known state before an object is returned to the pool. Note that because objects can be reused, this method may be called multiple times. Compare with `Cleanup()`.

CanBePooled method

`status = CanBePooled()`

`status`—a ^VAL value that indicates whether the object should be discarded or reused. Returns 0 if the object should be discarded; returns 1 if the object should be returned to the pool for reuse.

The `CanBePooled()` method is called after `Deactivate()`, and can be used to determine at runtime if an object can be re-used. For example, if `Deactivate()` encountered an error, `CanBePooled()` could indicate that the object should be discarded. Or, `Deactivate()` could be written to check how much effort is required to clean up an object before returning it to the pool. If the effort is excessive, and it would be more efficient to discard the object and create a new one, `CanBePooled()` would return 0.

`CanBePooled()` overrides the pool return setting in the application configuration file (see [page 17-8](#)).

Cleanup method

`Cleanup()`

`Cleanup()` is called by the object's `disconnect()` method. If the object's connection is shared, it is called after the final `disconnect()`. It can be used to do the final object cleanup, such as closing files. This method is called only when objects are going to be discarded (i.e., pool return is set to false or `CanBePooled()` returns 0). If the object is going to be reused, use the `Deactivate()` method to perform cleanup-type activities.

The `Cleanup()` method is also called when socket communication with the client is unexpectedly lost. When the pool is created, the `Cleanup()` method is automatically registered with the `XFPL_REGCLEANUP` routine on the server. (This routine must be in your SMC; see [page 1-34](#) for more information.) Then, if there is a fatal error that causes `xfServerPlus` to lose socket communication with the client, `xfServerPlus` calls the `Cleanup()` routine before shutting down.

Although `Cleanup()` and `Deactivate()` can be used for similar purposes, there are two fundamental differences between them:

- ▶ `Cleanup()` is called only for objects that will be discarded, while `Deactivate()` is called both for objects that will be discarded and for those that will be reused.
- ▶ `Cleanup()` is automatically registered so that `xfServerPlus` can call it when socket communication is lost.

Writing Code That Uses Pooled Objects

The code that you write when using pooling will look somewhat different than code for use without pooling, which was described in [“Using Your Synergy .NET Assembly” on page 17-9](#). Because the pooled objects already have a connection to *xyServerPlus*, you do not need to call the `connect()` method. The disconnection from *xyServerPlus* is also handled by the pool, so you do not need to call `disconnect()`.

You must add a reference in your Visual Studio project to `System.EnterpriseServices`. The *xyNetLink* .NET classes use this namespace.

If your client application is written in C#, when using pooling you should instantiate the object with a `using` statement as shown in the example below. Instantiating the object in this manner means that it exists only within the scope of the `using` statement. When the `using` statement ends, the object is released, and the `Deactivate()` and `CanBePooled()` methods are called. Failure to instantiate the object with a `using` statement can result in problems with the object being properly released and returned to the pool for reuse.

For example:

```
string userID = "MFranklin";
string password = "abc123";
try
{
    using (AppLogin userSess = new AppLogin())
    {
        userSess.login(userID, password);
    }
}
catch (Exception e)
{
    // Error handling code
}
```



If you are using ASP, instantiating the object within a `using` statement limits you to using the object on a single ASP page. If your application requires that the object have session scope (that is, you need to maintain state across several ASP pages), we do not recommend using pooling, as it is difficult to ensure that the object will be properly released, returned to the pool, and reused.

Calling Synergy Routines from .NET

Implementing Pooling

The following code shows the same example written in VB.NET. It uses the `Using` statement, which was introduced in VisualStudio 2005.

```
Dim userID As New String("MFranklin")
Dim password As New String("abc123")
Try
    Using userSess As New AppLogin()
        userSess.login(userID, password)
    Catch Ex As Exception
        ' Error handling code
    End Using
End Try
```

The following C# example shows how to use multiple copies of the same class in conjunction with pooling. (See [“Using Multiple Copies of the Same Class” on page 17-12](#) for more information about this feature.) In this example, our client application routes the user to a particular *xfServerPlus* port based upon a customer ID entered on the logon screen. We generated two instances of the *AppLogin* class, *AppLogin* and *AppLogin1*. We’ll use the generated C# interface (*IAppLogin*) to access methods in these classes so that all users can use the same client application, even though they are using different servers. As in the example above, we instantiate the object with the `using` statement.

```
// class fields
private string userID;
private string password;

// Method to set up the classes to create instances at each port
private void Button_Click(object sender,
    System.EventArgs e)
{
    // Class name consists of "Namespace.Class,Assembly"
    // Create instance for xfSP port 2356
    string className = "ABCComputers.AppLogin,ConsultIt";
    signOn(className);
    .
    .
    .
    // Create instance for xfSP port 4550
    className = "ABCComputers.AppLogin1,ConsultIt";
    signOn(className);
    .
    .
    .
}
```

```
// Method to create C# interface and use it with pooling
static void signOn(string className)
{
    using (IAppLogin userSess = getIAppLogin(className))
    {
        // call the Synergy method using the C# interface
        userSess.login(userID, password);
    }
}

// Method to create an instance of the correct class and cast it
// to the C# interface
static IAppLogin getIAppLogin(string className)
{
    // Dynamically create an instance
    Type tp = Type.GetType(className);
    Object obj = Activator.CreateInstance(tp);
    // Assign the object to the C# interface
    IAppLogin x = (IAppLogin)obj;
    return x;
}
```

Method Reference

Procedural Methods

This is a listing of the utility methods that are included in the procedural classes in your Synergy assembly.

connect()

```
public void connect()
```

Sends a request to *xfServerPlus* for a dedicated session and connects on the host and port defined in the application configuration file or, if those values are not defined in a config file, on the default host and port. See [“Connect to *xfServerPlus*” on page 17-10](#).

```
public void connect(string host, int port)
```

Sends a request to *xfServerPlus* for a dedicated session and connects to the specified host name on the specified port. Use this method to override at runtime the host and port specified in the configuration file (or the default host and port).

host—the name or IP address of the *xfServerPlus* machine

port—the port number that *xfServerPlus* is listening on. The port must be in the range 1024 through 65534.

```
public void connect(string host, int port, int minport,  
int maxport)
```



This method has been deprecated because the port number range feature is no longer supported. If *minport* and *maxport* are passed, they are ignored. We recommend that you use a `connect()` method that does not pass *minport* and *maxport*.

debugInit()

```
public void debugInit(ref string listen_ip, ref int listen_port)
```

Starts a connection to *xfServerPlus* so that you can manually connect an *xfServerPlus* session in debug mode. See [“Running an *xfServerPlus* Session in Debug Mode” on page 18-10](#).

listen_ip—returns the IP address, in hex, where the client is listening

listen_port—returns the port number where the client is listening

debugStart()

public void debugStart()

Completes the process of connecting in debug mode that was started by `debugInit()`. See [“Running an xfServerPlus Session in Debug Mode” on page 18-10](#).

disconnect()

public void disconnect()

Sends a message to close the *xfServerPlus* connection. See [“Disconnect from xfServerPlus” on page 17-12](#).

getConnect()

public object getConnect()

Gets the already-established connection of the specified object. See [“Connect to xfServerPlus” on page 17-10](#). This method is not included in assemblies built for pooling.

getUserString()

public string getUserString()

Returns the user string currently stored by `setUserString()`. See [“Writing to the xfServerPlus Log” on page 17-24](#).

setCallTimeout()

public void setCallTimeout(int *seconds*)

Sets the call time-out value, which measures the length of time that the client waits for a return from a remote call. See [“Setting a Call Time-Out” on page 17-24](#).

seconds—the number of seconds you want the *xfNetLink* .NET client to wait for a return from a call to *xfServerPlus*. The default is 1800 seconds (30 minutes).

setUserString()

public void setUserString(string *userString*)

Passes a user string to the *xfServerPlus* log. See [“Writing to the xfServerPlus Log” on page 17-24](#).

userString—the text that you want to write to the *xfServerPlus* log

shareConnect()

```
public void shareConnect(object connection)
```

Shares the specified connection. See [“Connect to xfServerPlus” on page 17-10](#). This method is not included in assemblies built for pooling.

connection—the object that represents the connection

Structure Methods

These methods are included in the structure classes in your assembly. See [“Using the Clone\(\) and Equals\(\) methods” on page 17-16](#) for more information and an example.

Clone()

```
public object Clone()
```

Returns an exact copy (including data) of the structure class on which it is called.

Equals()

```
public Boolean Equals(structClass instance)
```

Tests whether the current structure class (i.e., the one on which the method is called) and the passed structure class contain the same data. Returns true if they do and false if they do not.

instance—the instance of the structure class to be compared to the current class

DataTable Methods

These methods are included in the DataTable classes in your assembly. See [“Using DataTables” on page 17-18](#) for more information and examples.

AddRow()

```
public void AddRow(structClass instance)
```

Adds a new row after the last row in the DataTable.

instance—the instance of the structure class to add

```
public void AddRow(structClass instance, int rowPos)
```

Adds a new row at a specified location in the DataTable. The new row is added at the specified position, and the existing rows shift down one position.

instance—the instance of the structure class to add

rowPos—the position in the DataTable where you want to insert the row

GetRow()

```
public void GetRow(ref structClass instance, int rowPos)
```

Returns a row from a DataTable as an instance of a structure class.

instance—returns an instance of the structure class

rowPos—the position in the DataTable of the row to return

GetRows()

```
public ArrayList GetRows()
```

Returns all rows in the DataTable as an ArrayList of structures.

Error Handling and Troubleshooting in *xfNetLink* .NET

This chapter includes a table that lists the errors that can occur when using *xfNetLink* .NET Edition and what you can do to resolve them. It also includes information on troubleshooting, including how to turn on logging, run the *xfNetLink* .NET test program, and run a debug session.

Handling Errors

All *xfNetLink* .NET exceptions are thrown as `System.Exception`. In your code, you should catch `Exception`. The error text associated with the exception will tell you what the specific problem is. Check the table below for the likely cause and possible solutions for specific errors. Some errors may include additional text that was generated by the system or *xfServerPlus*. You may also see *xfServerPlus* status codes returned to the client. Refer to the table on [page 3-16](#).



When you get an error, the log file can provide information about what went wrong. See [“Using Client-Side Logging” on page 18-7](#). All errors are also written to the Windows application event log, regardless of whether client-side logging is turned on.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors		
Error message	Cause	What to do
Class <i>className</i> does not have a method named <i>methodName</i>	The specified method does not exist in the specified class.	Correct your client code. Consult the API documentation or the Object Browser in Visual Studio to determine the valid methods for this class.
Connection timed out	xfNetLink .NET timed out because of very slow start up.	Increase the connect time-out value. See “Creating and Editing Configuration Files” on page 17-5 .
Could not create new process	(Win, UNIX) Could not launch db s process.	Ensure that db s.exe exists on the xfServerPlus machine.
DBLDIR not set	The DBLDIR environment variable is not set on the server.	Set DBLDIR on the xfServerPlus machine.
Delimiter <i>delimiter</i> not found in packet	The packet is missing a delimiter. This message indicates a badly formed packet. The delimiter may be any of the following: #();;	Retry. The problem may be noise on the line or some other type of transmission error.
Expected DateTime type not found	A date or time data type other than DateTime was sent. xfNetLink .NET supports only the DateTime data type for dates.	Correct your client code to use a DateTime data type. See “xfNetLink .NET” in Appendix B .
Host or port not configured properly	The host and/or port setting have not been set up properly in the config file. Or, there is no config file, causing the program defaults to be used (localhost and 2356), and those values are invalid.	If you are using a config file, use the Synergy .NET Configuration utility to verify that the host and port are set correctly. See “Creating and Editing Configuration Files” on page 17-5 . If you are not using a config file, verify that xfServerPlus is running as localhost on port 2356.
Incompatible client version	The xfNetLink .NET version is incompatible with the xfServerPlus version.	Upgrade the client or the server to the higher version. Running a newer client with an older server is not a supported configuration.
Initialize method failed	(pooling only) The Initialize() method (defined in the SMC) failed on the xfServerPlus side.	Consult xfpl.log for details on why the method failed. It may be the case that the initialize time-out occurred before the method completed.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors (continued)		
Error message	Cause	What to do
Insufficient memory on server	(Win) Could not launch dfs process due to insufficient memory on the server.	Check the resources on your <i>xfServerPlus</i> machine.
Invalid date/time: <i>passed</i> Format: <i>expected</i>	Data in a date/time user-defined field (in a repository structure) is not in the expected format. The message will show you what was passed and the expected format.	Check the format of the data in your server-side code.
Invalid parameter number	The number of parameters passed to <i>xfServerPlus</i> does not match the number of parameters returned.	Check both your client-side and server-side code to verify that you are passing the correct number of parameters.
Invalid synergy type sent. Structure size passed and SMC size declared don't match.	The size of the parameter in the call doesn't match the size in the SMC. The error message includes the parameter number.	Check the definition of the method to see what size the parameter should be. Correct your code if necessary. If the error is in the SMC, update it and regenerate the assembly. If the parameter is a structure, the problem may be that the repository was updated, but the SMC was not. See "Verifying Repository Structure Sizes and Enumerations" on page 2-42.
Launch reply returned error	An error occurred in the response from <i>xfServerPlus</i> .	Refer to the error message for additional detail describing the problem. The message may include an additional error code returned by <i>xfServerPlus</i> ; if so, see the table on page 3-16 .
Logicals are not valid as logfile locations: <i>logicalName</i>	In the config file, the logfile setting was declared using a logical.	Change the value of the logfile setting in the config file to a path and filename. See "Creating and Editing Configuration Files" on page 17-5.
Login failed	(Win, UNIX) Invalid username or password used to start <i>xfServerPlus</i> .	Ensure that the username and password are correct for <i>xfServerPlus</i> . See "Running xfServerPlus" on page 3-3.
Max port is less than min port	In the config file, the maxport value is less than the minport value.	Redefine the port number range in the config file. See "Creating and Editing Configuration Files" on page 17-5.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors (continued)		
Error message	Cause	What to do
Method information must be passed to <code>callMethod</code>	Attributes for the method being called are not being passed to <code>xfNetLink</code> . (These attributes contain information about the method.)	This error occurs only when attribute information in the generated C# code has been altered or removed. If you know what was changed, you can correct the code. Otherwise, you must regenerate the classes (run gencs) and rebuild the assembly.
Min and max ports must be greater than 1024	In the config file, either the <code>minport</code> or the <code>maxport</code> setting is less than 1024.	Redefine the port number range in the config file. See "Creating and Editing Configuration Files" on page 17-5 .
Missing class attribute for <code>className</code>	An attribute for <code>className</code> is missing. (This attribute declares the length of a structure class.)	This error occurs only when attribute information in the generated C# code has been altered or removed. If you know what was changed, you can correct the code. Otherwise, you must regenerate the classes (run gencs) and rebuild the assembly.
Missing field attribute for <code>fieldName</code>	An attribute for <code>fieldName</code> is missing from a structure class. (This attribute contains the Synergy data type and length of the field.)	
Missing method attribute for <code>methodName</code>	An attribute for <code>methodName</code> is missing. (This attribute contains the name of the method in the SMC to call.)	
Missing parameter attribute for <code>parameterName</code>	An attribute for <code>parameterName</code> is missing. (This attribute contains the Synergy data type and length of the parameter.)	
No available ports in range	All ports in the specified range have been used.	Expand the range of ports set with <code>minport</code> and <code>maxport</code> in the config file. See "Creating and Editing Configuration Files" on page 17-5 .
No elements in array	An array was declared with no elements and then passed as a parameter.	Correct your client code to declare the elements in the array and, if it's an "in" or "in/out" parameter, set optional values for each element.
No runtime license	There is no Synergy Runtime license on the <code>xfServerPlus</code> machine.	Install a Runtime license on the <code>xfServerPlus</code> machine.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors (continued)		
Error message	Cause	What to do
Number of fields does not match	The number of fields returned by <i>xfServerPlus</i> does not match the number of fields in the structure class.	Check both your client-side and server-side code to verify that you are passing the correct number of fields for the structure.
Packet length error	The response message from <i>xfServerPlus</i> has an invalid packet length. This message indicates a badly formed packet.	Retry. The problem may be noise on the line or some other type of transmission error.
Parameter direction mismatch	The parameter direction (in, out, or in/out) is defined differently in the SMC than it is in the assembly.	Ensure that the parameter is defined correctly in the SMC and then regenerate the classes (run gencs) and rebuild the assembly.
Parameter number mismatch	The method call contains more or fewer parameters than the method supports.	Check your client code to verify that you are passing the correct number of parameters in your call.
Parameter type mismatch	The data type of a parameter sent back from <i>xfServerPlus</i> does not match the data type on the client side.	Check both your client-side and server-side code to determine where the error is. It may be the case that the server-side code was changed, but the SMC was not updated. If this is the case, you will need to update the SMC, regenerate the classes, and then rebuild the assembly.
Past end of packet	A badly formed packet caused <i>xfNetLink</i> to try to read more data than is in the packet.	Retry. The problem may be noise on the line or some other type of transmission error.
Returned array size does not match size of array in parameter	<i>xfServerPlus</i> returned an array that does not match the size of the array on the client side.	Check both your client-side and server-side code to verify that the array size is declared correctly.
Unexpected attribute type <i>attributeType</i>	The generated code contains an unsupported attribute type named <i>attributeType</i> .	This error occurs only when attribute information in the generated C# code has been altered or removed. If you know what was changed, you can correct the code. Otherwise, you must regenerate the classes (run gencs) and rebuild the assembly.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors (continued)		
Error message	Cause	What to do
Unexpected data type	The data in the parameter is not the correct type (e.g., alpha data in a decimal parameter).	Correct your client code. See “xfNetLink .NET” in Appendix B . Or, it may be the case that the parameter data type was entered incorrectly in the SMC, in which case you will need to correct the SMC, regenerate the classes, and rebuild the assembly.
Unexpected error	A server-side error occurred.	Note the error number; save your error log; contact Synergy/DE Developer Support.
Unexpected string in message	An unknown string of data was found in the packet. This message indicates a badly formed packet.	Retry. The problem may be noise on the line or some other type of transmission error.
Unsupported date/time format <i>format</i>	A structure in the repository uses an unsupported date/time format.	Correct the structure in the repository. Then, you must update the SMC, regenerate the classes, and rebuild the assembly. For information on supported data types, see “xfNetLink .NET” in Appendix B .
xfServerPlus license count exceeded	xfServerPlus has exceeded the number of available licenses.	Upgrade the number of licenses.
xfServerPlus license expired	The 14-day demo period or an extended demo period has expired.	Contact your Synergy/DE customer service representative.
xfServerPlus not licensed	There is no license for xfServerPlus.	Purchase a license.
xfServerPlus not supported by the running server	xfServerPlus is not running on the specified port.	Verify that xfServerPlus is running on the default port or the port specified in the config file or passed by the connect() method.
xfServerPlus version not supported	The version of xfServerPlus is not compatible with the version of xfNetLink for the current operation.	Upgrade xfServerPlus to the most current version.

Troubleshooting Techniques

Error messages don't always provide enough diagnostic information to solve a problem. In such cases, you can take advantage of the additional debugging options provided with *xfNetLink* and *xfServerPlus*: client-side logging, the *xfNetLink .NET* test program, and the ability to run an *xfServerPlus* session in debug mode.

You may also want to view the server-side logs (see [“Using Server-Side Logging” on page 3-29](#)) and run the test program included with *xfServerPlus* (see [“Testing *xfServerPlus*” on page 3-15](#)).

Using Client-Side Logging

You may want to use client-side logging when access to the server is inconvenient, or when you have used server-side logging and seen differences between what the client program is passing and what the server log shows is being received. All *xfNetLink .NET* client errors are written to the Windows application event log, regardless of whether logging is turned on. Client-side logging, however, enables you to log other types of information in addition to errors.

To turn on client-side logging, use the Synergy .NET Configuration utility to set the “logging” and “logfile” values in the application configuration file. By default, client-side logging produces a separate log file for each class that instantiates a connection to *xfServerPlus*. The log files are named with the specified (or default) log file name, plus a date/time stamp so that you can differentiate them. If you would prefer that all connections write to a single log file, use the Synergy .NET Configuration utility to turn on the “single log file” option. (See [“Creating and Editing Configuration Files” on page 17-5](#) for instructions on setting these values.)

One of the advantages of multi-file logging is improved performance. Because only one process is writing to the file, it can remain open. With single-file logging, the file must be opened and closed because different processes are writing to it, and this slows performance.

The information included in the log depends on the options you select in the Synergy .NET Configuration utility; see the table on [page 17-7](#) for the available options. For the sample log below, we turned on all logging. “00046X” is the identifier of the class object writing to this log. Note that if encryption is enabled, the log displays a string of 10 asterisks instead of the packet data for encrypted methods.

Error Handling and Troubleshooting in xfNetLink .NET

Troubleshooting Techniques

```
00046X:created instance of syntst
00046X:configuration settings
00046X: host = elmo
00046X: port = 2356
00046X: minport = 0
00046X: maxport = 0
00046X: connecttimeout = 120
00046X: initializettimeout = 30
00046X: poolreturn = False
00046X: logging = -1
00046X: logfile = c:\temp\xfnet.log
00046X: pooling = False
00046X: singlelogfile = on
00046X: xfNetLink .NET version = 9.3.1.1
*****
00046X:calling connect
00046X:host=elmo port=2356
00046X:connecting to xfServerPlus on host=elmo port=2356
00046X:sending launch request packet
00046X:packet length=56
00046X:8***&*****Q6**3*****
00046X:received launch response packet
00046X:packet length=52
00046X:4***&***,*****Y*=*****
00046X:received acknowledgement packet
00046X:packet length=53
00046X:5*****20-AUG-2009 12:03:32;00
00046X:sending second startup packet
00046X:packet length=43
00046X:+++N000010.1.3.51;0;*****
00046X:calling method function_one
00046X:Parameters:
00046X: p1=This is an alpha string that will likely be truncated
00046X: p2=12345
00046X: p3=9876543
00046X: p4=9876543
00046X:sending method call packet: 20-AUG-2009 12:03:32 PM
00046X:packet length=107
00046X:k***Nxfpl_tst1;4;AL50#This is an alpha string that will
likely be trunca;DE5#12345;ID7#9876543;DE7#9876543;
00046X:received method call reply packet: 20-AUG-2009 12:03:32 PM
00046X:packet length=19
00046X:***Rxfpl_tst1;000;
00046X:done calling method function_one
```

Testing xfNetLink .NET

The **xfTestnet.exe** program, distributed with *xfNetLink .NET*, can help you determine if your system is set up and working properly. The **xfTestnet** files are installed in the *xfNLNet\Examples* directory. **xfTestnet** requires that the .NET Framework SDK or the runtime (**dotnetredist.exe**) and *xfNetLink .NET* be installed on the machine running the test.

xfTestnet runs several tests, which send different types of data back and forth between the client and the Synergy server. This program makes calls to a test ELB or shared image named **xfpl_tst**, which is distributed with *xfServerPlus*. There are entries in the SMC for use by the test program. (These are the methods in the *xfTest* interface in the distributed SMC.) If the ELB or any of the methods are missing, the tests will fail.



If the methods in the *xfTest* interface are not present in your SMC, you can import them from the **defaultsmc.xml** file. See [“Importing and Exporting Methods” on page 2-39](#).

There is also a test program for *xfServerPlus*; see [“Testing xfServerPlus” on page 3-15](#).

► To run the *xfNetLink .NET* test program

1. Make sure *xfServerPlus* has been started on the server machine.
2. Make sure the .NET Framework and *xfNetLink .NET* have been installed on the client machine.
3. The default configuration settings (see the table on [page 17-3](#)) will be used unless you edit the application configuration file for the test program. If necessary, start the Synergy .NET Configuration utility and do the following:
 - Open the **xfTestnet.exe.config** file.
 - Edit the settings for host and port. If you need help using the Synergy .NET Configuration utility, see [“Creating and Editing Configuration Files” on page 17-5](#).
 - Save the configuration file and close the utility.
4. Run **xfTestnet.exe** from the *xfNLNET\Examples* directory.

5. If you don't want to use a shared connection, clear the checkbox. A shared connection will run faster.
6. Click the Run Tests button.

If the tests run successfully, you'll see a success message. If any tests are unsuccessful, an error message will display on the screen. The error message text should help you determine what needs to be done to correct the problem. Refer to the error message table beginning on [page 18-1](#). If you did not turn on logging, you may want to do so and run the test again. Then, if you cannot solve the problem, save the log file and contact Synergy/DE Developer Support.

Running an xfServerPlus Session in Debug Mode

During normal operation, xfServerPlus runs as a background process without support for console operations, complex user interfaces, or debugging. This improves efficiency and minimizes memory requirements. However, there may be times when you need to run the debugger on Synergy code in the ELBs that are being called from xfServerPlus. By manually connecting an xfServerPlus session to your .NET client application, you can run your Synergy server routines in debug mode so that you can uncover problems that are showing up as errors in your distributed application.



We recommend that you use the Telnet method for debugging if the operating system of your xfServerPlus machine is Windows or UNIX. See ["Debugging Remote Synergy Routines via Telnet"](#) on [page 3-42](#) for instructions.

Running in debug mode on Windows and UNIX

Use this procedure if the operating system of your xfServerPlus machine is Windows or UNIX.

If your SMC files or **xfpl.ini** file are not in the default location (DBLDIR), you will need to either move them to DBLDIR or set XFPL_SMCPATH and XFPL_INIPATH *in the environment* to point to the location of the files before

starting **xfpl.dbr** (step 4). (Note: When XFPL_SMCPATH and XFPL_INIPATH are set in the registry or **synrc**, they are read by **rsynd**. Since **rsynd** is bypassed when you run in debug mode, the registry/**synrc** settings do not get read.)

1. Use the `debugInit()` method to initiate a debug session. This method binds an IP address and port number for listening, and then returns the IP and port. You need to include code that displays the IP address (in hex) and port on the screen.

For example, in C#:

```
; declare variables
string listen_ip = "";
string listen_port = "";
; instantiate object and make call
AppLogin userSess = new AppLogin();
userSess.debugInit(listen_ip, listen_port);
; display hex IP and port
Console.WriteLine ("IP = "+ listen_ip +" Port = "+listen_port);
```

2. When the IP and port display on the screen, write them down. You'll need them in step 4. For example:

```
IP = 6F16212C
Port = 1082
```



Once the IP address and port display on the screen, you have a limited amount of time in which to manually start **xfServerPlus** in debug mode on the server machine, specify a breakpoint, and type "go". The default time-out for debug mode is 10 minutes. You can change this value by setting the connect time-out value in the Synergy .NET Configuration utility. See ["Creating and Editing Configuration Files" on page 17-5](#).

3. Use `debugStart()` to complete the connection process:

```
userSess.debugStart();
```

At this point, the client application has opened a socket and is waiting for the server to call it back.

4. Go to the machine running **xfServerPlus**, start **xfpl.dbr**, and pass the IP and port to **xfServerPlus**. Type the alpha characters in the IP address in uppercase.

```
dbr -d xfpl hexadecimal_ip listen_port
```

Error Handling and Troubleshooting in xfNetLink .NET

Troubleshooting Techniques

For example:

```
dbr -d xfp1 6F16212C 1082
```

xfServerPlus starts up in the debugger window.

5. Set an initial breakpoint in the **xfpl** program at the **XFPL_DEBUG_BREAK** routine. In the debugger enter

```
break xfp1_debug_break
```

and then enter

```
go
```

xfServerPlus is now connected to the client on the specified port. The server waits while the client program resumes and makes its first call. The program will then break at the **XFPL_DEBUG_BREAK** routine. This breakpoint occurs just after xfServerPlus has opened the ELB for the first method called by your application. (Note that any ELBs linked to this ELB will also be opened.) The ELB must be opened before you can set breakpoints in the routines within it.

6. If the Synergy routine you need to debug is in one of the opened ELBs, just specify a breakpoint in that routine. If the routine you want to debug is in a different (unopened) ELB, use the **OPENELB** debugger command to open that ELB. (You can also continue running your client application until the ELB is opened by xfServerPlus. However, because you set a breakpoint at **XFPL_DEBUG_BREAK**, it will break at each method call, so using the **OPENELB** command is more efficient.)



For general information about the Synergy debugger, see the “[Debugging Your Synergy Programs](#)” chapter of *Synergy Language Tools*. For details on the **OPENELB** command, see **OPENELB** in that same chapter.

Running in debug mode on OpenVMS

Use this procedure if the operating system of your xfServerPlus machine is OpenVMS.

1. Make sure xfServerPlus is running on an unused port. If necessary, restart it to ensure that it's using an unused port.
2. On the machine running xfServerPlus, enter

```
$ run DBLDIR:xfpld
```

You'll see output similar to the following:

```
*****
***  DEBUG 9.3.1   ***
BREAK AT 152 IN XFPL (LAUNCHER.DBL;6) ON ENTRY
%DBG-E-Could not open source file "LAUNCHER.DBL;1"
Dbg!Dbg>
*****
```



If you have created shared image logicals for the shared images used by *xfServerPlus*, you can skip [step 3](#). Instead, set a breakpoint for your shared image and routine as described in [step 6](#). You'll then be prompted for the port number ([step 4](#)). Once you start your client program ([step 5](#)), the debug session will break at the breakpoint you set.

3. Set an initial breakpoint in the *xfpl* program at the `XFPL_DEBUG_BREAK` routine. In the debugger enter

```
set break xfpl_debug_break
```

and then enter

```
go
```

4. When prompted, enter the port number that *xfServerPlus* is running on (from [step 1](#)).
5. Start your client application in the usual manner. After *xfNetLink* connects, the debug session will break at the `XFPL_DEBUG_BREAK` routine.
6. Set a breakpoint for your Synergy shared image and routine:

```
set break image/routine
```

(For details, see [BREAK](#) in the “Debugging Your Synergy Programs” chapter of *Synergy Language Tools*.)

Note that if you set a breakpoint at `XFPL_DEBUG_BREAK`, the debugger will break at `XFPL_DEBUG_BREAK` for each method call your client makes.



Although you do not need to use the `OPENELB` debugger command before setting the first breakpoint in your shared image, you may need to use it if your code does an `XSUBR` or `RCB_SETFNC` without specifying a shared image. For details on the `OPENELB` command, see [OPENELB](#) in the “Debugging Your Synergy Programs” chapter of *Synergy Language Tools*.

Appendices ▶

Configuration Settings

This appendix lists the configuration settings and environment variables that you can set for *xfServerPlus* and the *xfNetLink* clients.

xfServerPlus

When the *xfServerPlus* service starts, **rsynd** reads settings in the registry on Windows—or in the **synrc** file on UNIX and OpenVMS—and uses them to set up the environment for *xfServerPlus*. Each time an *xfServerPlus* session is started, **xfpl.dbr** reads the settings in the **xfpl.ini** file. Note that if you have set **SFWINIPATH** to point to the location of your **synergy.ini** file, the service runtime (**db.exe**) will read the **synergy.ini** file.

For best results, we recommend that you put the **XFPL_LOG** setting *first* in your **xfpl.ini** file, followed by the other logging settings and then the non-logging settings. This way, *xfServerPlus* knows immediately how—or whether—to create the log and, assuming logging is turned on, can then log any errors it encounters when reading the remaining settings.

xfpl.ini File Settings		
Use this setting	To specify this	For more information
XFPL_BASECHAN	The starting channel number used by <i>xfServerPlus</i> to open files.	“Specifying a Base Channel Number” on page 1-7
XFPL_COMPRESS	Repeated zeroes and spaces in data sent to and from COM, Java, and .NET clients should be compressed.	“Configuring Compression” on page 3-22
XFPL_LOGICAL ^a	Directories that ELBs are located in. If you used logicals in the Synergy Method Catalog, you must define them in xfpl.ini .	“Defining Logicals” on page 1-4
XFPL_LOG	Server-side logging is turned on.	XFPL_LOG on page 3-30

Configuration Settings

xfServerPlus

xfpl.ini File Settings (continued)		
Use this setting	To specify this	For more information
XFPL_LOGFILE	A filename for the log file.	XFPL_LOGFILE on page 3-30
XFPL_SINGLELOGFILE	All processes logged in a single file. The default is to produce a separate log file for each process.	XFPL_SINGLELOGFILE on page 3-31
XFPL_SESS_INFO	The level of session logging.	XFPL_SESS_INFO on page 3-31
XFPL_FUNC_INFO	The level of function logging.	XFPL_FUNC_INFO on page 3-34
XFPL_DEBUG	Debugging information should be written to the log.	XFPL_DEBUG on page 3-36

- a. On OpenVMS, we recommend setting logicals in **DBLDIR:SERVER_INIT.COM** rather than **xfpl.ini**. See [“Defining Logicals” on page 1-4](#).

Use the following environment variables to specify the locations of files that *xfServerPlus* uses. These environment variables should not be set in the **xfpl.ini** file; instead, set them as specified in the sections cited below.

Environment Variables		
Use this environment variable	To specify this	For more information
XFPL_INIPATH	Location of the xfpl.ini file. Required only if the file is not in DBLDIR.	“Setting the XFPL_INIPATH Environment Variable” on page 3-19
XFPL_SMCPATH	Location of the SMC files. Required only if the files are not in DBLDIR.	“Setting the XFPL_SMCPATH Environment Variable for xfServerPlus” on page 2-46

xfNetLink Synergy

xfNetLink Synergy on Windows uses settings in the **synergy.ini** file. These same settings are set as environment variables on UNIX and as logicals on OpenVMS.

synergy.ini File Settings		
Use this setting	To specify this	For more information
XF_REMOTE_HOST	Machine where xfServerPlus is running.	"Specifying the Host Name and Port Number" on page 5-3
XF_REMOTE_PORT	Port that xfServerPlus is running on.	
XF_RMT_TIMEOUT	Call time-out for regular session or debug.	"Call time-out" on page 5-5
XF_RMT_DBG_TIMEOUT	Session time-out for running in debug mode.	"Connect session time-out" on page 5-4
XF_RMTCONN_TIMEOUT	Session time-out for running in normal mode.	
XFNLS_LOGFILE	Name of file to log packets in.	"Specifying Debug Options" on page 5-6

xfNetLink Java

xfNetLink Java uses settings in the xfNetLink Java properties file (named **xfNetLink.ini** by default). When you use Java connection pooling, xfNetLink Java also uses the pooling properties file (**xfPool.properties**). Because these are Java properties files, the settings in them are case sensitive. For more information on these properties files, see [“Configuring the xfNetLink Java Properties File” on page 10-2](#) and [“Setting Up a Pooling Properties File” on page 10-24](#).

xfNetLink Java Properties File Settings		
Use this setting	To specify this	For more information
xf_RemoteHostName	Machine where xfServerPlus is running.	“Specifying the Host Name and Port Number” on page 10-6
xf_RemotePort	Port that xfServerPlus is running on.	
xf_LocalMinPort ^a	Lower limit of port number range for SynergyWebProxy to use to communicate with xfServerPlus.	“Specifying a Port Number Range” on page 10-7
xf_LocalMaxPort ^a	Upper limit of port number range for SynergyWebProxy to use to communicate with xfServerPlus.	
xf_DebugOutput	Information useful in debugging should be logged.	“Specifying Logging Options” on page 10-8
xf_LogFile	Name of file to write output to.	
xf_SessionRequestTimeout	Amount of time xfNetLink should wait for acknowledgment from the connection monitor in xfServerPlus.	“Request for session time-out” on page 10-9

xfNetLink Java Properties File Settings (continued)		
Use this setting	To specify this	For more information
xf_SessionConnectTimeout	Amount of time xfNetLink should wait for acknowledgment from the logic server in xfServerPlus when running in normal mode.	"Connect session time-out" on page 10-9
xf_DebugSessionConnectTimeout	Amount of time xfNetLink should wait for acknowledgment from the logic server in xfServerPlus when running in debug mode.	
xf_SessionLingerTimeout	Amount of time xfNetLink should wait for a return from a remote call.	"Call time-out" on page 10-10

- a. The port number range feature will be deprecated in a future version; we do not recommend using this setting.

Pooling Properties File Settings		
Use this setting	To specify this	For more information
minPool	The minimum number of connections to be maintained in the pool.	"Specifying the pool size" on page 10-26
maxPool	The maximum number of connections to be maintained in the pool.	
propertiesFile	The path and filename of the xfNetLink Java properties file to use. Required.	"Specifying the xfNetLink Java properties file to use" on page 10-27
poolReturn	A Boolean value that indicates whether the connection should be returned to the pool for reuse or discarded.	"Specifying whether connections should be returned to the pool" on page 10-27

Pooling Properties File Settings (continued)		
Use this setting	To specify this	For more information
poolLogFile	The filename that logging information should be written to.	“Specifying logging options” on page 10-28
poolLogLevel	The level of logging desired. Possible values are none, error, and all. If poolLogFile is specified and poolLogLevel is not specified, error level logging will take place.	
connectWaitTimeout	Number of seconds that the getConnection() method will wait for a connection from the pool.	“Specifying time-out values” on page 10-29
initializationTimeout	Number of seconds that xfNetLink should wait for a return from a remote call to xfServerPlus when the pool is being started. This value controls a special call time-out that applies only to the method specified with the initializationMethodID setting.	
initializationMethodID	Method ID of the Synergy method that will be called each time a new connection is added to the pool.	“Specifying the pooling support methods to call” on page 10-29
activationMethodID	Method ID of the Synergy method that will be called each time getConnection() is called.	
deactivationMethodID	Method ID of the Synergy method that will be called each time a connection is freed.	
poolableMethodID	Method ID of the Synergy method that will be called after the method specified by deactivationMethodID is called.	
cleanupMethodID	Method ID of the Synergy method that will be called each time a connection is discarded (i.e., not returned to the pool for reuse).	

xfNetLink COM

xfNetLink COM uses settings in the Windows registry, which are entered with the Synergy Type Library Configuration utility.

Windows Registry Settings for xfNetLink COM		
Use this setting	To specify this	For more information
xfhost	Machine where xfServerPlus is running.	"Using the Synergy Type Library Configuration Utility" on page 14-2
xfport	Port that xfServerPlus is running on.	
minport ^a	The lower end of the range of ports that you want the client to use to communicate with xfServerPlus.	
maxport ^a	The upper end of the range of ports that you want the client to use to communicate with xfServerPlus.	
logging	The type of information you want logged.	
logfile	Name of file you want to write logging information to.	
connect timeout	Amount of time xfNetLink should wait for an acknowledgment from the session started by the logic server in xfServerPlus.	
initialize timeout	(pooling only) Amount of time xfNetLink should wait for a return from a remote call to xfServerPlus when the pool is being started.	
pool return	(pooling only) Objects should be returned to the pool after they are used.	

a. The port number range feature will be deprecated in a future version; we do not recommend using this setting.

xfNetLink .NET

xfNetLink .NET uses settings in an application configuration file, which are entered with the Synergy .NET Configuration utility. See [“Using an Application Configuration File” on page 17-3](#) for general information on configuration files and whether you need to use one.

Configuration File Settings for xfNetLink .NET		
Use this setting	To specify this	For more information
host	Machine where xfServerPlus is running.	“Creating and Editing Configuration Files” on page 17-5
port	Port that xfServerPlus is running on.	
minport ^a	N/A	
maxport ^a	N/A	
logging	The type of information you want logged.	“Creating and Editing Configuration Files” on page 17-5 and “Using Client-Side Logging” on page 18-7
logfile	Name of file you want to write logging information to.	
single log file	Whether you want all sessions to share a common log file or want separate log files for each class that instantiates a connection to xfServerPlus.	
connect timeout	Amount of time xfNetLink should wait for an acknowledgment from the session started by the logic server in xfServerPlus.	“Creating and Editing Configuration Files” on page 17-5
initialize timeout	(pooling only) Amount of time xfNetLink should wait for a return from a remote call to xfServerPlus when the pool is being started.	
pool return	(pooling only) Objects should be returned to the pool after they are used.	

a. The port number range feature has been deprecated. This setting has no effect.

Data Type Mapping

This appendix includes data type mapping information for *xfNetLink* Java, *xfNetLink* COM, and *xfNetLink* .NET.

xfNetLink Java

Parameter and return value type mapping

The table below shows how parameter and return value data types are mapped from Synergy to Java when you generate Java class wrappers.

Note that for the primitive Java data types, “in” parameters are mapped as listed in the table, but “out” and “in/out” parameters actually use the corresponding holder classes. For example, an implied-decimal data type that is an “in” parameter is mapped to a `double`, while an implied-decimal data type that is an “out” or “in/out” parameter is mapped to a `DoubleHolder`. (See the Java `org.omg.CORBA` class for more information on holder classes.)

<i>xfNetLink</i> Java Parameter and Return Value Type Mapping		
Data type in SMC	Size in SMC	Java data type
$\wedge$ VAL ^a	N/A	int
Alpha	≤ 65,535	String (“in” parameters) StringBuffer (“out” and “in/out” parameters)
Binary (handle) ^b	N/A	ArrayList
Decimal (d)	1	byte
	2	short
	3 or 4	int
	≥ 5	long
Enumeration	N/A	Not supported

Data Type Mapping
xfNetLink Java

xfNetLink Java Parameter and Return Value Type Mapping (continued)		
Data type in SMC	Size in SMC	Java data type
Handle ^b	N/A	String ("in" parameters) StringBuffer ("out" and "in/out" parameters)
Implied-decimal (d.)	Any	double
Integer	1	byte
	2	short
	4	int
	8	long
System.String	N/A	Not supported

- a. Return value only.
- b. Parameter only.

Field type mapping

The table below shows how data types within repository structures are mapped to Java when you generate class wrappers. The structure itself becomes a class; this table shows what the data types for the properties of that class will be. The first column indicates the value in the Type field of the Field Definition window in Repository. The second column indicates the value in the Class field in that same window.

xfNetLink Java Field Type Mapping		
Field data type in Repository	Format	Java data type
Alpha	N/A	Mapping is the same as for parameters; see the table on page B-1 .
Binary	N/A	String
Boolean	N/A	byte
Date	YYMMDD YYYYMMDD YYJJJ YYYYJJJ	Date ^a
	YYPP YYYYPP	String
Decimal (with or without precision)	N/A	Mapping is the same as for parameters; see the table on page B-1 .
Enum	N/A	Not supported
Integer	N/A	Mapping is the same as for parameters; see the table on page B-1 .
Struct	N/A	N/A. Becomes a class (treated the same as a group).
Time	HHMM HHMMSS	Date
User	Any	String

- a. When a Synergy routine sends a zero date to xfNetLink Java, a Date object with a default date of 01/01/0001 is created. Your client application needs to test for this date to know that a zero date was sent. When xfNetLink Java sends a date of 01/01/0001 to a Synergy routine, a zero date is created.

Data translation

The table below shows how the SynergyType classes and the information in the SMC are used to translate the data sent by the SynergyWebProxy to xfServerPlus. This information can be useful if you are using the xfNetLink Java classes directly.

xfNetLink Java Data Translation				
Java data type	SynergyType class	Synergy data type in SMC	If the length in the SMC is...	Then...
byte short int long	synDec	Integer	1	Data is converted to an i1. An error occurs if the value sent is too big for the conversion.
			2	Data is converted to an i2. An error occurs if the value sent is too big for the conversion.
			4	Data is converted to an i4. An error occurs if the value sent is too big for the conversion.
			8	Data is converted to an i8. An error occurs if the value sent is too big for the conversion.
			None of the above, but the incoming number fits in the supported int length (L_ADDR_LEN)	Data is converted to an int of the maximum supported length (machine dependent).
			None of the above, and the incoming number does not fit in the supported int length (L_ADDR_LEN)	Error

xfNetLink Java Data Translation (continued)				
Java data type	SynergyType class	Synergy data type in SMC	If the length in the SMC is...	Then...
byte short long int	synDec	Decimal	Not specified and precision is not specified	Data is converted to a 'd' with the same length as the number being converted.
			≥ length of number being converted	Number is put into that value (right justified with leading zeros).
			< length of number being converted	Number is put into that value, significant digits are truncated, and no error is given.
		Implied decimal	Specified and precision is specified	The incoming number is processed this way: ▶ The decimal value of the string is aligned with the implied decimal of the 'd' (e.g., if you pass 1,000 to a d4.1, the value will be zero). ▶ If the string is longer than the integer portion, significant digits are truncated and no error given (e.g., if you pass 1,000 to a d3, the value will be zero).
		Alpha	Not specified	Number is converted to an alpha with the same length as the number being converted. Precision is ignored.
			≥ length of number being converted	Number is put into the alpha value (left-justified, trailing blanks are stripped).
			< length of number being converted	Number is put into the alpha value (left-justified), and the remainder of the number is truncated.

Data Type Mapping

xfNetLink Java

xfNetLink Java Data Translation (continued)				
Java data type	SynergyType class	Synergy data type in SMC	If the length in the SMC is...	Then...
float double	synImpDec	Integer	1	Data is converted to an i1. The value is rounded to the nearest whole number based on the first truncated digit.
			2	Data is converted to an i2. The value is rounded to the nearest whole number based on the first truncated digit.
			4	Data is converted to an i4. The value is rounded to the nearest whole number based on the first truncated digit.
			8	Data is converted to an i8. The value is rounded to the nearest whole number based on the first truncated digit.
			None of the above, but the incoming number fits in the supported int length (L_ADDR_LEN)	Data is converted to an int of the maximum supported length; else error. The value is rounded to the nearest whole number based on the first truncated digit.
		Decimal	Not specified and precision is not specified	Data is converted to a 'd' with the same length as the length of the string being converted. The value is rounded to the nearest whole number based on the first truncated digit.
			≥ length of string being converted	String is put into that value (right-justified with leading zeros). The value is rounded to the nearest whole number based on the first truncated digit.
			< length of string being converted	String is put into that value (right-justified), significant digits are truncated, and no error given. The value is rounded to the nearest whole number based on the first truncated digit.

xfNetLink Java Data Translation (continued)				
Java data type	SynergyType class	Synergy data type in SMC	If the length in the SMC is...	Then...
float double	synImpDec	Implied decimal	Specified and precision is specified	The string is processed this way: ▶ The decimal value of the string is aligned with the implied decimal of the 'd'. ▶ If the digits to the left of the decimal do not fit in the integer portion, significant digits are truncated and no error given. ▶ If the digits to the right of the decimal do not fit in the decimal portion, they are rounded based on the first truncated digit.
		Alpha Floating point representation of numbers	Not specified	Data is converted to an alpha with the same length as the length of the string being converted.
			≥ length of number being converted	Number is put into the alpha value (left justified, trailing blanks are stripped).
			< length of number being converted	Number is put into the alpha value (left-justified), and the remainder of the string is truncated.
String	synAlpha	Alpha	Not specified	Data is converted to an alpha with the same length as the string being converted.
			≥ length of string being converted	String is put into that value (left-justified, trailing blanks are stripped).
			< length of string being converted	String is put into that value (left-justified) and the remainder of the string is truncated.
	synHandle	Handle	N/A	A memory handle is created and the contents of the Java string become the contents of the handle.
ArrayList	synByteArray	Handle (binary)	N/A	A memory handle is created and the contents of the ArrayList become the contents of the handle.

xfNetLink COM

Parameter and return value type mapping

The table below shows how parameter and return value data types are mapped from Synergy to COM when the type library is created.

xfNetLink COM Parameter and Return Value Type Mapping		
Data type in SMC	Size in SMC	Vartype in TLB
^VAL ^a	N/A	VT_I2
Alpha	≤ 65,535	VT_BSTR
Binary (handle)	N/A	Not supported
Decimal (d)	1-4	VT_I2
	5-9	VT_I4
	>9	VT_R8
Enumeration	N/A	Not supported
Handle ^b	N/A	VT_BSTR
Implied-decimal (d.)	Any	VT_R8
Integer	1,2	VT_I2
	4	VT_I4
	8	VT_R8
System.String	N/A	Not supported

a. Return value only.

b. Parameter only.

Field type mapping

The table below shows how data types within repository structures are mapped to COM when the type library is created. The structure itself becomes a class; this table shows what the data types for the properties of that class will be. The first column indicates the value in the Type field of the Field Definition window in Repository. The second column indicates the value in the Class field in that same window.

xfNetLink COM Field Type Mapping		
Field data type in Repository	Format	Vartype in TLB
Alpha	N/A	VT_BSTR
Binary	N/A	VT_BSTR
Boolean	N/A	VT_I2
Date	YYMMDD YYYYMMDD YYJJJ YYYYJJJ	VT_DATE
	YYPP YYYYPP	VT_BSTR
Decimal (with or without precision)	N/A	Mapping is the same as for parameters; see the table on page B-8 .
Enum	N/A	Not supported
Integer	N/A	Mapping is the same as for parameters; see the table on page B-8 .
Struct	N/A	N/A. Becomes a class (treated the same as a group).
Time	HHMM HHMMSS	VT_DATE
User	Any	VT_BSTR

Type coercion

When a parameter is passed from your ASP or VB code to *xfNetLink* COM, if its data type does not match what is defined in the type library, it will be coerced to the correct type when possible. The exception is null parameters. An optional input parameter that is null will not be sent. A required “in” or “in/out” parameter that is null will be coerced to the correct type.



Input parameters that are required must be initialized. Uninitialized parameters cannot be coerced and will result in an error.

Coercion is deemed to be possible when the *vartype* describes a primitive type (not a date or time type) and the contents of the parameter can be converted in an unambiguous manner to the data type specified in the type library. When coercion is possible, the data is coerced to the appropriate *vartype* based upon the information in the TLB. When coercion is not possible, an error is thrown.

The table below shows the rules for type coercion. The first column indicates the expected *vartype* for the parameter, as defined in the TLB. The second column shows the circumstances under which a coercion to that type is possible.

xfNetLink COM Type Coercion	
Vartype in TLB	Coercion rules
VT_BSTR	▶ If parameter is numeric, it will be converted to a string representation of that value with leading and trailing zeroes suppressed (or “0” if zero). ▶ If parameter is empty, the string value will be “”.
VT_I2	▶ If parameter is numeric and the value fits in a VT_I2, it will be converted to a VT_I2. ▶ If parameter is a string that contains an int that will fit in a VT_I2, it will be converted to a VT_I2. ▶ If parameter is empty, the value will be 0.

xfNetLink COM Type Coercion (continued)	
Vartype in TLB	Coercion rules
VT_I4	▶ If parameter is numeric and the value fits in a VT_I4, it will be converted to a VT_I4. ▶ If parameter is a string that contains an int that will fit in a VT_I4, it will be converted to a VT_I4. ▶ If parameter is empty, the value will be 0.
VT_R8	▶ If parameter is numeric and the value fits in a VT_R8, it will be converted to a VT_R8. ▶ If parameter is a string that contains an int that will fit in a VT_R8, it will be converted to a VT_R8. ▶ If parameter is empty, the value will be 0.

xfNetLink .NET

Parameter and return value type mapping

The table below shows how parameter and return value data types are mapped from Synergy to C# when the classes are generated.

For decimal, implied-decimal, and integer data types, you can choose to coerce the data to a non-default type on the .NET side by selecting the desired data type in the Coerced type field when defining methods in the MDU. (See [page 2-27](#).) When coercing data to a non-default data type, you should take care to select a type that is suitable for the size of the data that will be placed in the parameter or return value. The data type in the “Default C# data type” column will be used when “Default” is specified for the coerced type in the MDU. (If you are using attributes instead of the MDU, see the description of the cType property on [page 2-10](#) for instructions on specifying non-default type mapping.)

xfNetLink .NET Parameter and Return Value Type Mapping		
Data type in SMC	Default C# data type	Non-default C# data types available
^VAL ^a	int	N/A
Alpha	string with trailing spaces trimmed	N/A
Binary (handle) ^b	byte array	N/A
Decimal (d)	length ≤ 9 = int length > 9 = long	byte, short, sbyte, ushort, uint, ulong, Boolean, DateTime, nullable DateTime
Enumeration	enumeration	N/A
Handle ^b	string	N/A
Implied-decimal (d.)	decimal	double, float
Integer	length < 8 = int length of 8 = long	byte, short, sbyte, ushort, uint, ulong, Boolean
System.String	string	N/A

- a. Return value only.
- b. Parameter only.

Field type mapping

The table below shows how data types within repository structures are mapped to C# data types when the classes are generated. The structure itself becomes a class; this table shows what the data types for the properties or fields of that class will be.

The first column indicates the value in the Type field of the Field Definition window in Repository. The second column indicates the value in the Class field and, where necessary, the value in the User data field in that same window.

Some data types can be coerced to a non-default type on the .NET side. To use a non-default type, select the desired data type in the Coerced type field of the Field Definition window in Repository. When coercing data to a non-default data type, you should take care to select a type that is suitable for the size of the data that will be placed in the property or field. For decimal (no precision) and integer data types, the data type in the “Default C# data type” column will be used when “Default” is specified for the coerced type in Repository.

xfNetLink .NET Field Type Mapping			
Field data type in Repository	Format	Default C# data type	Non-default C# data types available
Alpha	N/A	string with trailing spaces trimmed	N/A
Binary	N/A	byte array	You can convert binary fields to strings (which was the pre-9.3 behavior) by using the gens -nb option. See page 16-18 .
Boolean	N/A	Boolean	N/A
Date	YYMMDD YYYYMMDD YYJJ YYYYJJJ	DateTime ^a	nullable DateTime
	YYPP YYYYPP	string	N/A
Decimal (no precision)	N/A	length ≤ 9 = int length > 9 = long	byte, short, sbyte, ushort, uint, ulong, Boolean, decimal, nullable decimal
Decimal (with precision)	N/A	decimal	double, float, nullable decimal

Data Type Mapping

xfNetLink .NET

xfNetLink .NET Field Type Mapping (continued)			
Field data type in Repository	Format	Default C# data type	Non-default C# data types available
Enum	N/A	enumeration	N/A
Integer	N/A	length < 8 = int length of 8 = long	byte, short, sbyte, ushort, uint, ulong, Boolean
Struct	N/A	N/A. Becomes a class (treated the same as a group).	N/A
Time	HHMM HHMMSS	DateTime	nullable DateTime
User	Date with ^CLASS^=YYYYMMDDHHMISS in the User data field	DateTime ^a	nullable DateTime
	Date with any other value in the User data field	string with trailing spaces trimmed	N/A
	Alpha		
	Binary		
	Numeric		

- a. When a Synergy routine sends a zero DateTime to xfNetLink .NET, a .NET default date of 01/01/0001 is created. Your client application needs to test for this date to know that a zero date was sent. When xfNetLink .NET sends a date of 01/01/0001 to a Synergy routine, a zero date is created.

xfNetLink Synergy Sample Code

Client Application (sync.lt.dbl)

This code sample illustrates how to use RXSUBR and the RX_xxx routines in a Synergy client application. **Synclt.dbl** calls a subroutine named HELLO on the server.

The **sync.lt** program prompts you for the server name and port number, calls %RX_START_REMOTE to request a remote execution session, and then calls %RXSUBR to make the remote call. It calls the subroutine HELLO on the remote server, which returns the phrase “Hello, <your name>.” Then the program closes down the session with RX_SHUTDOWN_REMOTE. **Synclt** also includes error handling code that illustrates how to use RX_GET_HALTINFO and RX_GET_ERRINFO.

If you want to try running this program, see the instructions following the code sample. This code is included in the dbl\examples directory in your Synergy/DE distribution.

```
.main sync.lt

; Synergy client routine

.define CHAN          ,1                ; display channel
.define HELLO_MODULE  ,"hello_routine"  ; Remote procedure name

; Modify the following define statement for your system

.define NAME          ,"Mark"           ; Name to display

external function
    rc_api            ,^val

.include "DBLDIR:rxapi.def"

.define RX_ERR_DEF
record errinfo:rec
.include "DBLDIR:rxerr.def"
.undefine RX_ERR_DEF
```

xfNetLink Synergy Sample Code

Client Application (synclt.dbl)

```
.define RX_FATAL_DEF
record haltinfoec
.include "DBLDIR:rxerr.def"
.undefine RX_FATAL_DEF

.align
record
    netid          ,i4          ; network connection ID
    status         ,i4          ; return status
    syserr         ,i4          ; returned system error
    port          ,i4          ; port number

record
    aport         ,a4
    machine       ,a80
    message       ,a30          ; message text

.proc
xcall flags(4020, 1)
open(CHAN, 0:C, "TT:")
display(1, "Enter machine: ")
reads(1, machine, done)
display(1, "Enter port number: ")
reads(1, aport, done)
onerror done
port = aport
offerror

status = %RX_START_REMOTE(netid, machine, port) ; Start xfServerPlus
syserr = %syserr
if (status)
    begin
        writes(1, "Unable to connect to remote session:")
        writes(1, "          status = " + %string(status))
        writes(1, "          syserr = " + %string(syserr))
        goto done
    end

    call do_hello          ; Call subroutine

done,
    if (netid)
        xcall RX_SHUTDOWN_REMOTE(netid)    ; Stop xfServerPlus
    close(CHAN)
    stop

do_hello,          ; Subroutine to call the remote procedure and handle errors

    display(CHAN, $SCR_CLR(SCREEN), $SCR_MOV(2,20))
    clear message
```

```

    onerror ($ERR_XFHALT) handle_fatal, ($ERR_TIMEOUT) handle_timeout,
&    handle_other          ;trap fatal and timeout separately, then trap all others
    xcall rxsubr(netid, HELLO_MODULE, NAME, message)
    offerror
    display(CHAN, "**** " + %atrim(message) + " ****")
    return

; Error handling routines

handle_fatal,
    offerror
    writes(CHAN, "Fatal error trapped")
    xcall RX_GET_HALTINFO(netid, haltinfoec)
    call disp_halt
    goto done

handle_timeout,
    offerror
    writes(CHAN, "Timeout error trapped")
    xcall RX_GET_ERRINFO(netid, errinfoec)
    call disp_err
    goto done

handle_other,
    offerror
    writes(CHAN, "Error trapped")
    xcall RX_GET_ERRINFO(netid, errinfoec)
    call disp_err
    goto done

disp_halt,
    writes(CHAN, "Subroutine:      " +
&    %atrim(haltinfoec.rx_fatalerror.subroutine_name))
    writes(CHAN, "Error line #:    " +
&    %string(haltinfoec.rx_fatalerror.error_line_number))
    writes(CHAN, "Error #:          " + %string(haltinfoec.rx_fatalerror.error_num))
    writes(CHAN, "System Error #:    " + %string(haltinfoec.rx_fatalerror.error_num))
    writes(CHAN, "Program name:      " + %atrim(haltinfoec.rx_fatalerror.prog_name))
    writes(CHAN, "Error text:       " + %atrim(haltinfoec.rx_fatalerror.error_text))
    return

disp_err,
    writes(CHAN, "Method ID:         " + %atrim(errinfoec.rx_stderror.method_id))
    writes(CHAN, "# of errors:    " + %string(errinfoec.rx_stderror.num_of_errors))
    writes(CHAN, "Error #:          " + %string(errinfoec.rx_stderror.error_num))
    writes(CHAN, "Description:      " + %atrim(errinfoec.rx_stderror.description))
    writes(CHAN, "Clarification:    " + %atrim(errinfoec.rx_stderror.clarifying_desc))
    return
.end

```

Server-Side Code (HELLO Subroutine)

The HELLO subroutine is the remote routine (on the server) called by **synclt**. This code is included in the dbf\examples directory in your Synergy/DE distribution.

```
.subroutine hello
;Arguments
    a_name      ,a
    a_message    ,a

.define        HELLO      ,"Hello "
.define        NONAME     ,"No name passed "

.proc
    if (^passed(a_name)) then
        a_message = %atrim(HELLO + a_name)
    else
        a_message = NONAME
    xreturn
.end
```

Running the Hello Program

1. Create an ELB or shared image named **hello.elb** containing the HELLO subroutine. Put the ELB on your server machine (the machine that *xfServerPlus* is running on).
2. Start the Method Definition Utility and create a method in the SMC for the HELLO subroutine. (See [“Using the MDU to Define Synergy Methods”](#) on page 2-22 for instructions.) Include the following information in your MDU entry:

Method name = hello_routine
Method ID = hello_routine (this is copied from the method name)
Routine name = hello
ELB/shared image name = DBLDIR:hello (change the logical if necessary)
Return type = No return value

The subroutine has two parameters, name and message. Set them up as follows:

Parameter name = name
Data type = Alpha
Length = 20
Data passed = In

Pass by = Descriptor
Required

Parameter name = message
Data type = Alpha
Length = 30
Data passed = In/Out
Pass by = Descriptor
Required

3. If *xfServerPlus* is not already running, start it on the server machine. See [“Running *xfServerPlus*” on page 3-3](#) for details.
4. Put **synclt.dbl** on the client machine and edit the NAME identifier.
5. Compile and link **synclt.dbl**.
6. Run **synclt.dbr**. You’ll be prompted to enter the machine name and port where *xfServerPlus* is running.

***x/f*NetLink Java Sample Code**

Client Application (hello.jsp and hello.java)

The code samples in this appendix illustrate how to create a program using either JavaServer Pages or Java. Both programs call a subroutine named HELLO on the server.

The **hello.jsp** program uses the `jsp:useBean` action to load a JavaBean created using the *x/f*NetLink Java component generation tools. It sets the host and port, calls the `connect()` method to make a connection to *x/f*ServerPlus, and then calls the `hello()` method, passing two parameters. The program returns the results and then calls the `disconnect()` method to close the connection. The **hello.jsp** page is called from the **hello.html** page, which includes a table that displays the results to the screen. The code for **errorpage.jsp** is also included below: **hello.jsp** sets **errorpage.jsp** as the page to go to if it encounters a Java exception.

The **hello.java** program creates a new `JavaTest` object, sets the host and port, and calls the `connect()` method to make the connection to *x/f*ServerPlus. It then calls the `hello()` method, passing two parameters, and displays the returned results to the screen. The connection is closed by the `disconnect()` method.

If you want to try running these programs, see the instructions following the code sample. These sample files are included in the Examples directory in your *x/f*NetLink Java distribution.

hello.jsp

```
<%@ page errorPage="errorpage.jsp" %>
<jsp:useBean id="helloexm" scope="session" class="JavaTest.JavaTest" />
<html>
<head>
<title>Hello Example </title>
</head>
<body>
<center>
<%
    String name = new String("");
    StringBuffer message = new StringBuffer("");
```

xfNetLink Java Sample Code

Client Application (hello.jsp and hello.java)

```
int rtnval = 0;
String errmsg = "";
name = request.getParameter("UserName");

// Enter xfpl name or IP address and port number
helloxm.setxfHost("hostIP");
helloxm.setxfPort(2356);

// Connect to xfServerPlus and call hello method
helloxm.connect();
helloxm.hello(name, message);

out.println("<br>");
out.println("The hello example returned message = ");
out.println(message);
out.println("<br>");

// shut down xfpl connection
helloxm.disconnect();
%>
</body>
</html>
```

hello.html

```
<html>
<head> </head>
<body>
<form action=hello.jsp method=post>
<p>
<table align=center bgcolor=#cccccc border=0 cellPadding=1
cellspacing=0 width=75%>
  <tr>
    <td colspan=2>
      <div align=center><strong><em> <font color=#0000ff face=""
        size=5>Hello Example!</font></em></strong></div>
      <br>
    </td>
  </tr>
  <tr>
    <td align=right>name:</td>
    <td><INPUT id=text1 name=UserName MAXLENGTH=10 size=10></td>
    <br>
  </tr>
  <tr>
    <td align=left> &nbsp;   </td>
    <td align=left> &nbsp;   </td>
    <br>
  </tr>
  <tr>
    <td colspan=2><br><hr width=95%></td>
```



```
</tr>
<tr>
  <td align=left> &nbsp; </td>
  <td align=left><br>
    <input type="submit" value=" Login " id=submit1 name=runLogin
      align=center style="HEIGHT: 67px; TOP: 55px; WIDTH: 73px">
    <br>
  </td>
</tr>
</table></p>
</form>
</body>
</html>
```

errorpage.jsp

```
<html>
<body text="red">
<%@ page isErrorPage="true" %>
<!-- Use the implicit exception object, which holds a -->
<!-- reference to the thrown exception. -->
The errorpage - Error: <%= exception.getMessage() %> has been reported.
</body>
</html>
```

hello.java

```
import java.io.*;
import JavaTest.*;
import org.omg.CORBA.*;
import Synergex.util.*;

public class hello
{
  static JavaTest tst = new JavaTest()
  public static void main(String argv[])
  {
    try
    {
      hello test = new hello();

      // Open the xfpl connection
      tst.setxfHost("hostIP");
      tst.setxfPort(2356);
      tst.connect();
      String name = new String("World");
      StringBuffer message = new StringBuffer("");

      // Make the call
      tst.hello(name, message);
      System.out.println("message = " + message);
    }
  }
}
```

xfNetLink Java Sample Code

Server-Side Code (HELLO Subroutine)

```
        // Close down the xfp1 connection
        tst.disconnect();
    } catch (xfJCWException e) {
        e.printStackTrace(System.err);
    }
}
}
```

Server-Side Code (HELLO Subroutine)

The HELLO subroutine is the remote routine (on the server) called by **hello.jsp** and **hello.java**. The **hello.dbl** file is included in the **dbl\examples** directory in your Synergy/DE distribution.

```
.subroutine hello
;Arguments
    a_name      ,a
    a_message   ,a
.define        HELLO      ,"Hello "
.define        NONAME     ,"No name passed "
.proc
    if (^passed(a_name)) then
        a_message = %atrim(HELLO + a_name)
    else
        a_message = NONAME
    xreturn
.end
```

Running the Hello Program

1. Create an ELB or shared image named **hello.elb** containing the HELLO subroutine. The HELLO subroutine is located in the **dbl\examples** directory. Put the ELB on your *xy*ServerPlus machine.
2. Use the Method Definition Utility to add the HELLO subroutine to the SMC. (See [“Using the MDU to Define Synergy Methods” on page 2-22](#) for instructions.) Include the following information in your MDU entry:

Method name = hello_routine

Interface name = JavaTest

Method ID = hello_routine (this is copied from the method name)

Routine name = hello

ELB/shared image name = DBLDIR:hello (change the logical if necessary)

Return type = No return value

The subroutine has two parameters, name and message. Set them up as follows:

Parameter name = name

Data type = Alpha

Length = 20

Data passed = In

Pass by = Descriptor

Required

Parameter name = message

Data type = Alpha

Length = 30

Data passed = In/Out

Pass by = Descriptor

Required

3. Using Workbench or the command line utilities, create a Java JAR file named JavaTest that includes the JavaTest interface. See [“Creating a Java JAR File in Workbench” on page 9-4](#) or [“Creating a Java JAR File from the Command Line” on page 9-8](#) for instructions.
4. If xfServerPlus is not already running, start it on the server machine. See [“Running xfServerPlus” on page 3-3](#) for details.
5. Deploy and run the client program.
 - ▶ If you’re running the JSP application, do the following:
 - ▶ Put the JAR file, JSP files, and HTML file in the location required by your servlet container.
 - ▶ Verify that your web server and servlet container are running.
 - ▶ Start the web browser on the client machine and load the **hello.html** page.
 - ▶ If you’re running the Java application, do the following:
 - ▶ Compile the **hello.java** file, and then put it and the JAR file in a location where Java can find them.
 - ▶ Set your classpath.
 - ▶ Run the program.

See [“Deploying Your Distributed Application” on page 10-1](#) if you need help with deployment.

Active Server Pages (ASP). A web page that combines HTML and programming code written in Visual Basic Script or JScript®. When a browser requests an Active Server Page, the web server executes the embedded program, allowing the web page to interact with databases and other programs.

API. Application program interface.

assembly. A collection of files, which is the primary unit of deployment with the .NET Framework. An assembly is self-describing. It is used to build applications (but is not an application itself).

attributes. A feature of Synergy Language that enable you to automate the population and maintenance of the Synergy Method Catalog. There are two attributes, `xfMethod` and `xfParameter`.

class. A prototype for an object. A class defines the variables and methods common to all objects in that class.

classpath. An ordered list of directories and JAR files that specifies the location of class files used by Java applications.

COM. Component Object Model. The binary, interoperability standard developed by Microsoft that allows component objects developed in different languages to call one another. COM objects can be accessed by any COM-compliant application.

COM+. An enhanced version of Microsoft's Component Object Model. Among other improvements, COM+ handles many resource management tasks that developers previously had to program. COM+ enables you to implement pooling for `xfNetLink COM` and `xfNetLink .NET`.

COM+ application. Not an application in the traditional sense, but rather a set of administrative data that contains information about a collection of components. To implement pooling for `xfNetLink COM` or `xfNetLink .NET`, you must create a COM+ application and then add components to it.

component. See [Synergy component](#).

db12xml. The utility used to parse the attribute information in Synergy code and output an XML file that can be imported into the SMC.

exception. In Java or .NET, an error.

GAC. Global assembly cache. A directory structure in which a .NET application will look for assemblies. Assemblies placed in the GAC are considered “public”, enabling a single copy of the assembly to be used by multiple applications.

genxml. A utility that generates an XML file from method definitions in the SMC and structure definitions in the repository.

host. Refers to the machine on which *x/ServerPlus* is running.

HTTP. Hypertext Transfer Protocol. The client/server TCP/IP protocol used on the World Wide Web for transferring HTML documents.

instantiate. The process used to create an instance of a class (i.e., an object). A class is a prototype; when you instantiate a class, you create an object with the attributes of that class.

JAR file. A Java archive file. A JAR file is a collection of compressed Java class files, similar to a ZIP file. JAR files provide a way to distribute numerous class files together. The classes in a JAR file can be used by a JSP or Java application without unpacking the archive.

Java. An interpreted language used to write applications, applets, and JavaServer Pages.

Java application. A stand-alone program written in Java that can run outside a browser.

JavaServer Pages (JSP). A web page that combines HTML and code written in Java. When a browser requests a JavaServer Page, the web server executes the embedded code, allowing the web page to interact with databases and other programs.

Java Runtime Environment (JRE). The runtime for Java applications. The JRE includes the JVM, the Java core classes, and supporting files.

Java Virtual Machine (JVM). The Java interpreter. A web browser must contain the JVM (i.e., must be Java-enabled or Java-compliant) to run applets or JavaServer Pages.

MDU. Method Definition Utility. The application used to add, modify, and delete information in the Synergy Method Catalog. The MDU also has facilities for importing and exporting methods from the SMC.

method ID. A unique, 31-character value in the Synergy Method Catalog. *xfServerPlus* uses this value to look up the routine to call. On *xfNetLink Synergy* and *xfNetLink Java* (without class wrappers), this value is also used in the client code to reference the Synergy routine.

method name. A 50-character value that you create to reference the Synergy routine. On *xfNetLink Java* (with class wrappers), *xfNetLink COM*, and *xfNetLink .NET*, this value is used in the client code to invoke the Synergy routine.

modularization. A programming technique that requires code to be an isolated functional unit with a well-defined, published interface (i.e., an argument list).

native image. An assembly that has been pre-compiled for the particular platform it is running on. An assembly installed as a native image is installed in the GAC with the *ngen.exe* utility.

net ID. See [network connection ID](#).

network connection ID. The ID returned by `%RX_START_REMOTE`. The network connection ID is a handle to a memory structure containing information that describes the location and state of the *xfServerPlus* session that was created. Used by *xfNetLink Synergy*.

procedural class. A class that has methods. Procedural classes are created from the interfaces in the SMC. Compare with [structure class](#).

procedural interface. An interface that has methods. Procedural interfaces are created from the interfaces in the SMC. This term is used with *xfNetLink COM*. Compare with [structure interface](#).

proxy. A server that substitutes for another server. It intercepts messages and routes them where they need to go. Proxies are usually used for security reasons. For example, in *xfNetLink Java*, the *SynergyWebProxy* receives requests from Java and negotiates with *xfServerPlus*.

RCB. Routine call block. See below.

routine call block. A block of memory that contains the information necessary—routine name, ELB, and arguments—to make a routine call. You can use a routine call block to make remote calls with *xfNetLink Synergy*.

"set" methods. Methods included in the generated JAR file, which can be used to set properties used by *xfNetLink Java*, such as the host name and port, log filename, and time-out values.

SMC. Synergy Method Catalog. Identifies the Synergy routines that you have prepared for remote calling. The SMC includes information such as the function or subroutine name, the ELB or shared image it is stored in, and the type and length of its parameters.

strong name key file. A file used to sign a .NET assembly, which usually includes a public and a private key that are subsequently embedded into the assembly.

structure class. A class that has only properties, which represent fields in a structure. Structure classes are built from structures in the repository, which are referenced in the SMC. Compare with [procedural class](#).

structure collection. An array of structures with a variable number of elements.

structure interface. An interface that has only properties, which represent fields in a structure. Structure classes are built from structures in the repository, which are referenced in the SMC. This term is used with *xy*NetLink COM. Compare with [procedural interface](#).

Synergy assembly. Refers to a .NET assembly that references Synergy methods.

Synergy COM component. Refers to a type library that references Synergy methods. Also referred to as a Synergy type library.

Synergy component. A general term that can refer to a Synergy JAR file, type library, or assembly.

Synergy method. A Synergy Language function or subroutine that has been made available for use with *xy*NetLink. Synergy methods must be contained in an ELB or shared image and must be defined in the Synergy Method Catalog.

SynergyType. Any of several *xy*NetLink Java classes that are used to package Java data for passing to *xy*ServerPlus.

SynergyWebProxy. An *xy*NetLink Java class that is instantiated in order to establish a connection between *xy*NetLink and *xy*ServerPlus and to make calls to Synergy routines via *xy*ServerPlus.

TLB. Type library. A file that describes interfaces (methods and properties) exposed by objects, and which allows developers to use these interfaces. TLBs are part of Microsoft's implementation of COM.

utility methods. Refers to the standard methods that are included in every Synergy JAR file, type library, and assembly. These methods enable you to perform utility functions such as connecting, setting a time-out value, and running a debug session.

Variant. The default data type of all variables used in Visual Basic and Active Server Pages. A `Variant` variable can hold any other data type (integer, string, double, etc.).

vartype. The data type held in a `Variant` variable.

XML. Extensible Markup Language. XML is a syntax, developed by the World Wide Web Consortium, for creating your own markup language. It is a simple text stream that contains text and markup codes (tags).

A

- Activate() method
 - xfNetLink COM 14-23
 - xfNetLink .NET 17-33
- activation method (Java) 10-33
- AddRow() method (.NET) 17-40
- alternate interface names 1-8
 - xfNetLink COM 13-7, 13-12
 - xfNetLink Java 9-6, 9-9
 - xfNetLink .NET 16-9, 16-14
- API
 - xfNetLink Synergy 7-1 to 7-28
 - xfServerPlus 1-30 to 1-36
- application configuration file 17-3 to 17-8
 - creating 17-5 to 17-8
 - general information 17-3 to 17-4
 - naming 17-4
 - overriding settings at runtime 17-4
 - program defaults 17-3
 - using with Visual Studio project 17-5
- application, deploying. *See* client application, deploying; server application, deploying
- arguments. *See* data; parameters
- ArrayDescriptor class (Java) 12-24
- ArrayList
 - passing with xfNetLink .NET 1-16
 - synArrayList class (Java) 12-36
 - synByteArray class (Java) 12-37 to 12-38
 - System.Collections.ArrayList class 1-16
 - using to pass binary data (Java) 10-17 to 10-18, 12-37 to 12-38
 - using to return structure collection
 - parameter 1-14 to 1-16
- arrays
 - defining in SMC 2-33
 - larger than 64K 1-17, 6-3
 - multi-dimensional 2-33
 - passing (Java) 12-32 to 12-35
 - pseudo 2-33
 - real 2-33
 - support for in structures passed as parameters 1-8

- support for in xfNetLink 2-33
- within repository structures (Java) 10-16
- See also* ArrayList; parameters; synArray; SynProxyMixedTypeArrayException

assemblies

- configuration file 17-3 to 17-8
- creating from command line 16-13 to 16-21
- creating in Workbench 16-5 to 16-11
- deploying on client 17-1 to 17-2
- excluding structure fields from 1-10
- signing 16-7, 16-8, 16-17, 16-25
- target .NET version 16-10
- using in an application 17-9 to 17-25
- See also* C# classes; xfNetLink .NET

AssemblyInfo.cs file

- creation of 16-11, 16-16
- editing 16-26

assignElement() method (Java) 12-34

- attributes 2-3 to 2-19
 - advantages of 2-4
 - dbl2xml utility 2-3, 2-5
 - examples 2-16
 - summary table 2-6
- xfMethod 2-8 to 2-12
- xfParameter 2-12 to 2-16

B

BadSynPacketException 11-2, 12-7

- base channel number 1-7

batch file

- xfNetLink Java 9-13
- xfNetLink .NET 16-20

binary data, passing

- general information 1-18 to 1-19
- xfNetLink Java 10-17 to 10-18, 12-37 to 12-38
- xfNetLink .NET 17-22 to 17-23

binary fields in structures (.NET) 16-18

- byte array, conversion from binary structure field 16-18

byte array, passing

- xfNetLink Java 10-17 to 10-18, 12-37 to 12-38
- xfNetLink .NET 17-22 to 17-23

Index

C

C

- C# classes 16-5 to 16-24
 - alternate names for 16-9, 16-14
 - converting binary fields 16-18
 - creating project for in Workbench 16-6 to 16-9
 - documentation for 16-7, 16-18, 16-27 to 16-29
 - editing 16-26
 - generating from command line 16-13 to 16-19
 - generating from Workbench 16-10 to 16-11
 - generating properties vs. fields 16-8, 16-18
 - multiple copies of 17-12 to 17-13
 - namespace, specifying 16-7, 16-17
 - understanding 16-22 to 16-24
 - See also* assemblies; xfNetLink .NET
- C# interfaces 16-22, 17-13, 17-36
- call time-out
 - xfNetLink COM 14-4, 14-12
 - xfNetLink Java 10-10, 10-19, 10-26
 - xfNetLink .NET 17-7, 17-24
 - xfNetLink Synergy 7-15
 - xfServerPlus 1-31
- calling Synergy routines
 - from COM client 14-6 to 14-10
 - from Java client 10-11 to 10-15
 - from .NET client 17-9 to 17-25
 - from Synergy client 6-1 to 6-3
- CanBePooled() method
 - xfNetLink COM 14-24
 - xfNetLink .NET 17-34
- “Cannot load random state” error 3-24
- catalog. *See* Synergy Method Catalog
- certificate file for encryption 3-25
- chaining 1-3
- Changed property (.NET) 17-16
- “Channel is in use” error 1-7, 1-33
- channels
 - specifying for SMC files 1-7
 - when using U_START routine 1-6
 - when using XFPL_LOG routine 1-33
- classpath, setting 9-3
- cleanup method
 - Java pooling 10-34
 - registering with xfServerPlus 1-34 to 1-36
- Cleanup() method
 - registering with xfServerPlus 1-34 to 1-36
 - xfNetLink COM 14-24
 - xfNetLink .NET 17-34
- client application, deploying
 - xfNetLink COM 14-1
 - xfNetLink Java 10-1 to 10-2
 - xfNetLink .NET 17-1 to 17-2
 - xfNetLink Synergy 5-1
- client-side logging. *See* logging
- Clone() method (.NET) 17-16, 17-40
- closing connection to xfServerPlus
 - xfNetLink COM 14-10
 - xfNetLink Java 10-15, 12-14
 - xfNetLink .NET 17-12
 - xfNetLink Synergy 7-17
- coercing data types. *See* type coercion
- collection. *See* structure collection parameter
- collectionType property 2-15
- COM component project, creating 13-5 to 13-7
- COM pooling 14-14 to 14-26
 - changing pool configuration 14-21
 - connection pooling 14-14
 - creating poolable objects 14-25
 - debugging remote routines and 3-41, 3-42
 - initialize time-out setting 14-4
 - object pooling 14-14
 - overview 14-14 to 14-15
 - pool return setting 14-5
 - pool size 14-19
 - reusing connections and objects 14-15
 - running application when pooling disabled 14-26
 - setting up 14-17 to 14-21
 - support methods 14-22 to 14-24
 - writing code that uses pooled objects 14-25
- comments
 - in C# source files. *See* C# classes: documentation for
 - in Java source files. *See* Javadoc
 - in pooling properties file 10-25
 - in Synergy source files 2-19
 - in synrc file 2-48, 3-20
 - in xfNetLink properties file 10-4
- comparing structure data (.NET) 17-16 to 17-18

- Component Information dialog box (Workbench)
 - COM project 13-5 to 13-7
 - Java project 9-4 to 9-7
 - .NET project 16-6 to 16-9
- components. *See* assemblies; JAR files; type libraries
- compression of data 3-22
- configuration file. *See* application configuration file
- connect time-out
 - xfNetLink COM 14-4
 - xfNetLink Java 10-9
 - xfNetLink .NET 17-7
 - xfNetLink Synergy 5-4, 7-19
- connect() method
 - xfNetLink COM 14-8, 14-27
 - xfNetLink Java 10-13, 10-35
 - xfNetLink .NET 17-10, 17-38
- ConnectException 11-2
- connection pooling. *See* COM pooling; Java pooling
- connection to xfServerPlus, closing
 - xfNetLink COM 14-10
 - xfNetLink Java 10-15, 12-14
 - xfNetLink .NET 17-12
 - xfNetLink Synergy 7-17
- converting data. *See* data type mapping
- cType property
 - method 2-10
 - parameter 2-14

D

- D_NO_GLOBAL_DATA 1-6
- data
 - accessing remotely from xfServerPlus 3-45 to 3-49
 - global 1-3, 1-6
 - larger than 64K 1-12 to 1-14
 - packaging into SynergyType objects
 - (Java) 12-42 to 12-43
 - packed 7-24
 - supported types 2-32
 - variable length 1-12 to 1-17, 6-3
 - See also* data type mapping; parameters
- data compression 3-22
- data type conversion. *See* data type mapping
- data type mapping
 - xfNetLink COM B-8 to B-9
 - xfNetLink Java 12-43, B-1 to B-7
 - xfNetLink .NET B-12 to B-14
- dataTable property 2-16
- DataTables (.NET) 16-23, 17-18 to 17-21
 - column caption 17-19
 - column names 17-19
 - defining in MDU 2-33
 - defining with attribute 2-16
 - examples 17-20
 - methods 17-20, 17-40
 - overview 16-23
 - primary key 17-20
- DateTime (.NET) 2-27, 2-33, B-13, B-14
- dbl2xml utility 2-3, 2-5
 - See also* Synergy Language Tools
- dbs.exe 1-5, 3-29, 3-40
 - See also* Synergy Language Tools
- Deactivate() method
 - xfNetLink COM 14-23
 - xfNetLink .NET 17-33
- deactivation method (Java) 10-33
- debug constructor 12-11, 12-12
- debug mode, running xfServerPlus in 3-40
 - from COM client 15-12 to 15-15
 - from Java client 11-11 to 11-15
 - from .NET client 18-10 to 18-13
 - from Synergy client 6-5 to 6-9
- debugging 3-40 to 3-44
 - debugInit() method. *See* debugInit() method
 - debugStart() method. *See* debugStart() method
 - overview 3-40
 - %RX_DEBUG_INIT function 7-4
 - %RX_DEBUG_START function 7-5 to 7-6
 - setting XFPL_DEBUG 3-36 to 3-38
 - SynergyWebProxy constructor for 12-11, 12-12
 - test skeletons, generating 1-20 to 1-29
 - with Telnet 3-41 to 3-44
 - xfNetLink COM 15-9 to 15-15
 - xfNetLink Java 11-7 to 11-15
 - xfNetLink .NET 18-7 to 18-13
 - xfNetLink Synergy 6-5 to 6-10
 - See also* errors; logging; testing; xfServerPlus log
- debugInit() method
 - xfNetLink COM 14-27, 15-13
 - xfNetLink Java 10-35, 11-12
 - xfNetLink .NET 17-38, 18-11

Index

E

- debugStart() method
 - xfNetLink COM 14-27, 15-13
 - xfNetLink Java 10-35, 11-12
 - xfNetLink NET 17-39, 18-11
 - default port, xfServerPlus 3-3
 - defaultsmc.xml file 2-39
 - delayed signing of assemblies 16-8, 16-17, 16-25
 - deploying applications. *See* client application, deploying;
server application, deploying
 - disconnect() method
 - xfNetLink COM 14-10, 14-27
 - xfNetLink Java 10-15, 10-35
 - xfNetLink .NET 17-12, 17-39
 - disconnecting from xfServerPlus
 - xfNetLink COM 14-10
 - xfNetLink Java 10-15, 12-14
 - xfNetLink .NET 17-12
 - xfNetLink Synergy 7-17
 - dllhost.exe.config file 17-4
 - documentation, creating
 - adding comments to Synergy source files 2-19
 - for .NET assemblies 16-7, 16-18, 16-27 to 16-29
 - Javadoc 9-5, 9-13, 9-16 to 9-18
- ### E
- early binding 14-8
 - elb property 2-9
 - ELBs 1-3 to 1-5
 - calling other ELBs 1-3
 - closing 1-31
 - defining in MDU 2-26
 - defining logicals for 1-4, 2-55
 - defining with attribute 2-9
 - linked 1-3
 - restrictions regarding 1-3
 - searching for in MDU 2-37
 - xfServerPlus time-out and 1-31
 - encrypt property 2-12
 - encryption 3-23 to 3-28
 - and xfNetLink .NET logging 18-7
 - and xfNetLink Synergy logging 6-9
 - and xfServerPlus logging 3-23
 - certificate file 3-25
 - client machine set-up 3-26
 - disabling 3-23
 - specifying data for slave encryption 3-28
 - xfServerPlus machine set-up 3-24
 - enumerations 17-21
 - defining in MDU 2-26, 2-30
 - environment variables
 - OPENSSLIB 3-25
 - SYNCSCOPT 16-11, 16-20
 - SYNSSL_RAND 3-24
 - SYNSSLIB 3-25
 - XF_REMOTE_HOST 5-3
 - XF_REMOTE_PORT 5-3
 - XF_RMT_DBG_TIMEOUT 5-4
 - XF_RMT_TIMEOUT 5-5
 - XF_RMTCONN_TIMEOUT 5-4
 - XFNLS_LOGFILE 5-6
 - XFPL_DBR. *See* *Environment Variables and System Options*
 - XFPL_INIPATH 3-19 to 3-21
 - XFPL_SMCPATH 2-46 to 2-49
 - Equals() method (.NET) 17-16, 17-40
 - \$ERR_CHNUSE error 1-7
 - \$ERR_FILOPT error 1-6
 - \$ERR_XFNOCALL error 7-3
 - \$ERR_XFNOINIT error 7-5
 - error handling. *See* errors
 - error log. *See* logging; xfServerPlus log
 - error tag 13-6, 13-13
 - errors
 - “Channel is in use” 1-7, 1-33
 - \$ERR_CHNUSE 1-7
 - \$ERR_FILOPT 1-6
 - \$ERR_XFNOCALL 7-3
 - \$ERR_XFNOINIT 7-5
 - handling (general information) 1-19
 - “Invalid operation for file type” 1-6
 - signaled by %RX_START_REMOTE 7-20 to 7-22
 - signaled by %RXSUBR 7-26 to 7-27
 - xfNetLink COM 15-1 to 15-8
 - xfNetLink Java 11-1 to 11-6
 - xfNetLink .NET 18-1 to 18-6
 - xfNetLink Synergy 6-3 to 6-4, 7-20 to 7-22,
7-26 to 7-27
 - xfServerPlus 3-16, 3-39 to 3-40
 - See also* debugging; exceptions

exceptions

- BadSynPacketException 11-2, 12-7
- ConnectException 11-2
- handling for xfnNetLink Java 11-1 to 11-6
- InterruptedIOException 11-3
- IOException 11-3
- SynProxyBadPropsException 11-3, 12-17
- SynProxyBadTypeException 11-3, 12-49
- SynProxyException 12-18
- SynProxyIncompatibleServerException 11-3, 12-19
- SynProxyInvalidStructException 11-3, 12-50
- SynProxyMixedTypeArrayException 11-3, 12-51
- SynProxyNetException 11-4, 12-3
- xfJCWException 11-4, 12-57
- xfLaunchFatalException 11-4, 12-20 to 12-21
- xfPoolException 11-5, 12-58
- xfRemoteExecException 11-5, 12-22 to 12-23
- xfServerNackException 11-6, 12-4 to 12-5
- xfServerPlusUnavailableException 11-6, 12-6
- See also* errors
- “Excluded by web” flag 1-10
- exporting methods from SMC 2-39 to 2-41

F

- fatal errors. *See* errors; exceptions
- FieldDescriptor class (Java) 12-25
- fields vs. properties, generating for .NET structure members 16-8, 16-18
- firewall
 - using with minport and maxport settings
 - xfNetLink COM 14-5
 - xfNetLink Java 10-7
 - when debugging via Telnet 3-41
- format property
 - method 2-12
 - parameter 2-14
- functions. *See* Synergy routines

G

- GAC 17-1, 17-2
- gens utility 16-16 to 16-19
- genjava utility 9-12 to 9-13
- gensxml utility. *See* genxml utility
- gensyn utility 1-24 to 1-25
- gentlb utility 13-15

genxml utility

- test skeletons 1-23
- xfNetLink COM 13-11 to 13-14
- xfNetLink Java 9-9 to 9-11
- xfNetLink .NET 16-13 to 16-16
- getConnection() method
 - xfNetLink COM 14-8, 14-27
 - xfNetLink Java 10-13, 10-35
 - xfNetLink .NET 17-10, 17-39
- getElement() method (Java) 12-34
- getInstance() method (Java) 12-55
- getMessage() method (Java) 12-4, 12-19, 12-20, 12-22
- GetRow() method (.NET) 17-41
- GetRows() method (.NET) 17-41
- getUserString() method
 - xfNetLink COM 14-12, 14-28
 - xfNetLink Java 10-20, 10-36
 - xfNetLink .NET 17-24, 17-39
- getxfHost() method (Java) 10-36
- getxfLogfile() method (Java) 10-36
- getxfMaxport() method (Java) 10-36
- getxfMinport() method (Java) 10-36
- getxfPort() method (Java) 10-36
- global data 1-3
 - excluding from tools.def file 1-6

H

- handle. *See* memory handle
- help, obtaining xv
- host name, defining on client
 - xfNetLink COM 14-5
 - xfNetLink Java 10-6, 10-13, 10-36
 - xfNetLink .NET 17-7
 - xfNetLink Synergy 5-3, 7-18
- host port number, defining on client
 - xfNetLink COM 14-5
 - xfNetLink Java 10-6, 10-13, 10-36
 - xfNetLink .NET 17-8
 - xfNetLink Synergy 5-3, 7-18

I

- id property 2-9
- ImpDecDescriptor class (Java) 12-26
- importing methods to SMC 2-39 to 2-41
- initialization method (Java) 10-33

Index

J

- initialize time-out
 - xfNetLink COM 14-4
 - xfNetLink Java 10-26, 10-29
 - xfNetLink .NET 17-7
- Initialize() method
 - xfNetLink COM 14-23
 - xfNetLink .NET 17-33
- instantiating SynergyType objects 12-42
- interface property 2-8
- interfaces, naming 2-24
 - See also* alternate interface names
- InterruptedException 11-3
- “Invalid operation for file type error” 1-6
- IOException 11-3
- IP address. *See* host name, defining on client
- IStructManager interface 12-27

J

- JAR files
 - creating from command line 9-8 to 9-14
 - creating in Workbench 9-4 to 9-8
 - deploying on client 10-1
 - excluding structure fields from 1-10
 - using classes in 10-11 to 10-15
 - See also* Java class wrappers; xfnljav.jar file
- Java class wrappers 9-4 to 9-16
 - alternate names for 9-6, 9-9
 - creating documentation for 9-5, 9-13, 9-16 to 9-18
 - creating project for in Workbench 9-4 to 9-7
 - editing the generated files 9-16
 - generating from command line 9-9 to 9-13
 - generating from Workbench 9-7
 - package name, specifying 9-5, 9-12
 - understanding the generated files 9-14 to 9-16
 - See also* JAR files; xfNetLink Java
- Java classes 12-1 to 12-58
 - See also specific Java classes*
- Java component project, creating 9-4 to 9-7
- Java JAR file. *See* JAR files
- Java pooling 10-20 to 10-34
 - debugging remote routines and 3-41, 3-42
 - error logging 10-21, 10-28
 - overview 10-20 to 10-21
 - pool ID 10-24
 - pool maintenance 10-31 to 10-32
 - pool size 10-26

- properties file 10-24 to 10-30
- returning pool to minimum size 10-31
- reusing connections 10-21, 10-27, 10-34
- shutting down the pool 10-32, 12-56
- support methods 10-29, 10-32 to 10-34
- time-outs 10-29
- updating properties files 10-31
- usePool() method 10-23, 10-39
- using multiple pools 10-24, 10-25
- writing code that uses pooled connections 10-22 to 10-24
- See also* pooling properties file

Javadoc

- deploying 10-2
- generating 9-5, 9-13, 9-16 to 9-18

JavaScript

JavaServer Pages

- code sample D-1 to D-5
- using with xfNetLink Java 8-1 to 8-3

JDK, supported versions 8-1

K

- KEEPALIVE timer value 1-31
- key file. *See* strong name key file

L

- late binding 14-8
- length property
 - method 2-10
 - parameter 2-13
- license requirements for xfServerPlus xiv
- “log on as a batch job” user right, error if not set 7-21, 11-6
- “log on locally” user right 3-3
- logging
 - xfNetLink COM 14-4, 15-9 to 15-10
 - xfNetLink Java 10-8, 10-28, 11-7 to 11-9
 - xfNetLink .NET 17-7, 18-7
 - xfNetLink Synergy 5-6, 6-9 to 6-10
 - xfServerPlus 3-29
 - See also* xfServerPlus log
- logicals, defining for ELBs 1-4, 2-55
 - See also* environment variables

M

maximum port. *See* port number range

maxport. *See* port number range

MDU. *See* Method Definition Utility

mdu.dbr 2-50 to 2-54

See also Method Definition Utility

memory handle

using to pass binary data

general information 1-18 to 1-19

xfNetLink Java 10-17 to 10-18

xfNetLink .NET 17-22 to 17-23

using to pass large parameters 1-12 to 1-13

using to pass variable-length parameters 1-12 to 1-13

using to return structure collection 1-14 to 1-16

Method Definition Utility 2-22 to 2-38

changing method ID 2-28

command line syntax 2-50 to 2-54

creating new methods 2-22 to 2-34

date last updated field 2-35

defining parameters 2-28 to 2-34

defining structure parameters 2-30

deleting data 2-36

exporting methods 2-39 to 2-41, 2-51

importing methods 2-39 to 2-41, 2-51, 2-52

modifying entries 2-35 to 2-36

remote SMC, using with 2-46, 2-50

resequencing parameters 2-35

searching for data in 2-37

specifying alternate location for SMC files 2-45

specifying method ID 2-28

verifying structure sizes 2-42

See also Synergy Method Catalog

method IDs

defining in MDU 2-28

defining with attribute 2-9

passing to %RXSUBR 7-23

passing to xcall() method 12-15

vs. method name when making remote calls 2-2

See also Method Definition Utility

methods

included in Java JAR file 10-35 to 10-39

included in .NET assembly 17-38 to 17-41

included in type library 14-27 to 14-28

pooling support

xfNetLink COM 14-22 to 14-24

xfNetLink Java 10-32 to 10-34

xfNetLink .NET 17-32 to 17-34

Synergy. *See* Method Definition Utility; Synergy

Method Catalog

minimum port. *See* port number range

minport. *See* port number range

modular code 1-1

multiple copies of C# classes 17-12 to 17-13

multiple SMCs 2-43 to 2-49

multiple xfpl.ini files 3-18

N

name property

method 2-9

parameter 2-13

namespace, specifying 16-7, 16-17

NDoc application 16-29

.NET component project, creating 16-6 to 16-9

.NET Framework

installing redistributable 17-2

regsvcs utility 17-29

sdkvars.bat file 16-10, 16-20

setting up environment 16-10, 16-20

sn.exe 16-25

supported version 16-1

System.Data.DataTable class 17-18

target version for generated assembly 16-10

.NET pooling 17-26 to 17-37

ASP and 17-35

building assembly for pooling 16-20

changing pool configuration 17-31

checking the pool status 17-31

creating poolable objects 17-27

debugging remote routines and 3-41, 3-42

initialize time-out setting 17-7

overview 17-26 to 17-27

pool return setting 17-8

pool size 17-30

reusing objects 17-27

setting up 17-28 to 17-31

starting the pool 17-31

support methods 17-32 to 17-34

writing code that uses pooled objects 17-35

Index

O

network connection ID 6-2, 7-18
 invalid 7-28
nullable DateTime (.NET) B-13
 defining in MDU 2-27, 2-33
 defining with attribute 2-12, 2-15
nullable property
 method 2-12
 parameter 2-15

O

object pooling. *See* COM pooling; .NET pooling
OpenSSL. *See* encryption
OPENSSLIB environment variable 3-25
 See also *Environment Variables and System Options*
OpenVMS
 servstat program 3-13, 3-14
 unsupported routines 1-4
Original property (.NET) 17-17
overlays in repository structures 1-9 to 1-11

P

package name, specifying 9-5, 9-12
packed data 7-24
packets, viewing
 on client side. *See* logging
 on server side. *See* xfServerPlus log
parameters
 ArrayList class 1-16, 2-33
 arrays 2-33
 arrays larger than 64K 1-17
 binary 1-18 to 1-19
 coercing data type 2-33, B-12 to B-14
 DataTable 16-23, 17-18 to 17-21
 larger than 64K 1-12 to 1-14, 1-17
 xfNetLink Java 12-44 to 12-45
 xfNetLink Synergy 6-3
 maximum number allowed 2-28
 null (Java) 12-48
 optional 2-34, 10-14, 17-11
 packed 7-24
 passing structures as 1-8 to 1-11
 ^REF 2-34, 7-24
 required 2-34, 10-14, 17-11
 sequence of 2-35

String class 1-13
structure collection 1-14 to 1-16, 2-33
supported by %RXSUBR 7-24
supported data types 2-32
System.Collections.ArrayList class 1-16, 2-33
System.String class 1-13
type mapping B-1 to B-14
^VAL 2-34, 7-24
variable length 1-12 to 1-17, 6-3
 See also Method Definition Utility
.pem file 3-5, 3-25
poolable method (Java) 10-34
pooling properties file 10-24 to 10-30
 comments in 10-25
 multiple pools 10-24, 10-25
 naming 10-24
 placement of 10-24
 pool ID 10-24
 rereading while pool is active 10-31
 sample 10-30
 settings 10-25 to 10-30
 activationMethodID 10-29
 cleanupMethodID 10-29
 connectWaitTimeout 10-29
 deactivationMethodID 10-29
 initializationMethodID 10-29
 initializationTimeout 10-29
 maxPool 10-26
 minPool 10-26
 poolableMethodID 10-29
 poolLogFile 10-28
 poolLogLevel 10-28
 poolReturn 10-27
 propertiesFile 10-27
pooling. *See* COM pooling; Java pooling; .NET pooling
port number range
 xfNetLink COM 14-5
 xfNetLink Java
 specifying in constructor 12-8
 specifying in properties file 10-7
 specifying with set method 10-13, 10-38
xfNetLink .NET 17-8

- port number, xfServerPlus
 - default 3-3
 - setting
 - OpenVMS 3-12, 3-13
 - UNIX 3-10
 - Windows 3-4, 3-6
 - See also* host port number, defining on client
 - precision property
 - method 2-10
 - parameter 2-13
 - Professional Series Workbench. *See* Workbench
 - properties file. *See* pooling properties file; xfNetLink Java properties file
 - properties vs. fields, generating for .NET structure
 - members 16-8, 16-18
 - protocol version 12-19
 - pseudo arrays 2-33
- R**
- range of ports for client. *See* port number range
 - RCB. *See* routine call block
 - rcbid 7-16
 - rd.log file 3-44
 - read-only properties (.NET) 16-8, 16-18, 17-14
 - real arrays 2-33
 - ^REF parameters
 - defining in MDU 2-34
 - passing to %RXSUBR 7-24
 - registering
 - assembly in GAC 17-1, 17-2
 - cleanup method with
 - XFPL_REGCLEANUP 1-34 to 1-36
 - type library 13-8, 13-16, 14-2
 - xfServerPlus on Windows 3-6
 - regsvcs utility 17-29
 - regtlb utility 13-16
 - remote data access from xfServerPlus for 3-45 to 3-49
 - repository fields. *See* structures
 - repository files, specifying location
 - assembly 16-8, 16-14
 - JAR file 9-5, 9-10
 - test skeletons 1-22
 - type library 13-6, 13-12
 - repository structures. *See* structures
 - request for session time-out
 - xfNetLink COM 14-4
 - xfNetLink Java 10-9
 - xfNetLink .NET 17-7
 - xfNetLink Synergy 5-4
 - resetPoolProperties() method (Java) 10-31, 12-55
 - returnToMinimum() method (Java) 10-31, 12-56
 - routine call block 6-3, 7-16
 - routines. *See* Synergy routines; Synergy/DE routines
 - RPSDAT. *See* repository files, specifying location
 - rpsmain.ism. *See* repository files, specifying location
 - RPSMFIL. *See* repository files, specifying location
 - rpstext.ism. *See* repository files, specifying location
 - RPSTFIL. *See* repository files, specifying location
 - rsynd program
 - starting. *See* starting xfServerPlus
 - stopping. *See* stopping xfServerPlus
 - syntax. *See* *Installation Configuration Guide*
 - running xfServerPlus 3-3 to 3-14
 - OpenVMS 3-12 to 3-14
 - UNIX 3-9 to 3-10
 - Windows 3-3 to 3-7
 - %RX_CONTINUE function 7-2
 - %RX_DEBUG_INIT function 6-6, 7-4
 - %RX_DEBUG_START function 6-6, 7-5 to 7-6
 - RX_GET_ERRINFO subroutine 7-7 to 7-8
 - RX_GET_HALTINFO subroutine 7-9 to 7-10
 - %RX_RMT_ENDIAN function 7-11
 - %RX_RMT_INTSIZE function 7-12
 - %RX_RMT_OS function 7-13
 - %RX_RMT_SYSINFO function 7-14
 - %RX_RMT_TIMEOUT function 7-15
 - RX_SETRMTFNC subroutine 7-16
 - RX_SHUTDOWN_REMOTE subroutine 7-17
 - %RX_START_REMOTE function 7-18 to 7-22
 - status codes 7-20 to 7-22
 - rxapi.def file 6-1, 7-18
 - rxerr.def file 6-1, 7-7
 - %RXSUBR routine 7-23 to 7-27
 - runtime errors signaled by 7-26 to 7-27
 - using to make remote call 6-1 to 6-2

Index

S

S

Sandcastle application 16-29
ScalarDescriptor class (Java) 12-28
sdkvars.bat file 16-10, 16-20
sendShutdownMessage() method (Java) 12-14
server application, deploying 3-44
service runtime 1-5, 3-29, 3-40
 See also Synergy Language Tools
servstat program 3-14
 displaying xfServerPlus status 3-13
 purging free pool 3-14
“set” methods (Java)
 compared with properties file 10-5 to 10-6
 using 10-13
 See also individual methods
SET_XFPL_TIMEOUT subroutine 1-31
setCallTimeout() method (.NET) 17-39
setruser 3-9, 3-46, 3-48
setUserString() method
 xfNetLink COM 14-12, 14-28
 xfNetLink Java 10-20, 10-36
 xfNetLink .NET 17-24, 17-39
setxfCallTimeout() method
 xfNetLink COM 14-12, 14-28
 xfNetLink Java 10-19, 10-37
setxfHost() method (Java) 10-37
setxfLogfile() method (Java) 10-37
setxfLogging() method (Java) 10-37
setxfMaxport() method (Java) 10-38
setxfMinport() method (Java) 10-38
setxfPort() method (Java) 10-38
SFWINIPATH environment variable 3-40
shareConnect() method
 xfNetLink COM 14-8, 14-28
 xfNetLink Java 10-13, 10-38
 xfNetLink .NET 17-10, 17-40
shared images. *See* ELBs
SHELL routine 1-4
shutdown() method (Java) 10-32, 12-56
SMC. *See* Synergy Method Catalog
smc_elb.exe 2-55 to 2-58
SMC/ELB Comparison utility 2-55 to 2-58
sn.exe 16-25
SPAWN routine 1-4

starting xfServerPlus
 OpenVMS 3-12 to 3-14
 UNIX 3-9 to 3-10
 Windows 3-3 to 3-7
status codes
 %RX_START_REMOTE 7-20 to 7-22
 xfServerPlus 3-16
stopping xfServerPlus
 OpenVMS 3-14
 UNIX 3-11
 Windows 3-8
strong name key file 16-7, 16-17, 16-25
struct data type fields in repository 1-8
StructDescriptor class (Java) 12-29
structfields 1-8, 2-7
 examples 2-16, 2-19
structure classes, using
 xfNetLink COM 14-8, 14-10 to 14-11
 xfNetLink Java 10-15 to 10-17
 xfNetLink .NET 17-14 to 17-18
structure collection parameter 1-14 to 1-16
 defining in MDU 2-33
 defining with attribute 2-15
structure property 2-15
structures
 binary fields (.NET) 16-18
 comparing data in (.NET) 17-16 to 17-18
 defining in MDU 2-30
 excluding fields from 1-10
 field type mapping
 xfNetLink COM B-9
 xfNetLink Java B-3
 xfNetLink .NET B-13
 overlays, handling 1-9 to 1-11
 passing as parameters 1-8 to 1-11
 properties vs. fields for .NET 16-8, 16-18
 read-only fields in 17-14
 returning collection of 1-14 to 1-16
 supported types 1-8
 verifying sizes in SMC 2-42
support, obtaining technical xv
SWPConnect class (Java) 12-53
SWPManager class (Java) 12-55
 using methods in 10-22, 10-31 to 10-32
synAlpha class (Java) 12-30 to 12-31

- synArray class (Java) 12-32 to 12-35
 - creating empty arrays 12-34
- synArrayList class (Java) 12-36
- synByteArray class (Java) 12-37 to 12-38
- SYNCSOPT environment variable 16-11, 16-20
- synDec class (Java) 12-39 to 12-41
- Synergy code. *See* Synergy routines; Synergy server code
- Synergy components. *See* assemblies; JAR files; type libraries
- Synergy Configuration Program
 - setting XFPL_INIPATH 3-19 to 3-20
 - setting XFPL_SMCPATH 2-46 to 2-47
 - starting xfServerPlus 3-3 to 3-6
 - stopping xfServerPlus 3-8
- Synergy Language data types
 - defining in MDU 2-26, 2-31
 - mapping to C# types B-12 to B-14
 - mapping to COM types B-8 to B-9
 - mapping to Java types 12-43, B-1 to B-7
 - supported 2-32
- Synergy Method Catalog 2-1 to 2-58
 - accessing files on remote machine 2-46, 2-50
 - creating new files 2-43 to 2-44
 - default location 2-43
 - exporting methods 2-39 to 2-41
 - importing methods 2-39 to 2-41
 - overview 2-1
 - populating. *See* attributes; Method Definition Utility
 - specifying when creating component
 - assembly 16-9, 16-14
 - JAR file 9-6, 9-10
 - type library 13-7, 13-12
 - specifying when creating test skeletons 1-23
 - updating 2-39 to 2-41
 - using logicals in 1-4, 2-55
 - using multiple 2-43 to 2-49
 - verifying entries against ELB 2-55 to 2-58
 - verifying structure sizes 2-42
 - See also* attributes; Method Definition Utility
- Synergy .NET Configuration utility 17-5 to 17-8, 17-28
- Synergy routines
 - calling from client
 - xfNetLink COM 14-6 to 14-10
 - xfNetLink Java 8-5, 10-11 to 10-15
 - xfNetLink .NET 17-9 to 17-25
 - xfNetLink Synergy 6-1 to 6-3
 - debugging 3-40 to 3-44
 - defining for use with xfNetLink 2-1 to 2-37
 - searching for in MDU 2-37
 - See also* Synergy/DE routines; Synergy server code
- Synergy server code
 - debugging 3-40 to 3-44
 - defining routines for use with xfNetLink 2-1 to 2-37
 - ELB restrictions 1-3
 - enclosing in ELBs 1-3
 - modularizing 1-1
 - preparing for remote access 1-1 to 1-29
 - testing 1-20 to 1-29
 - UI Toolkit routines and 1-5 to 1-6
 - user interface elements in 1-5
- Synergy Type Library Configuration utility 14-2 to 14-3, 14-17
- Synergy/DE routines
 - UI Toolkit routines 1-5 to 1-6
 - unsupported 1-3, 1-4
 - See also* Synergy routines
- synergy.ini file
 - configuring for xfNetLink Synergy 5-2 to 5-6
 - read by dbs 3-40
- SynergyType class (Java) 12-42 to 12-43
 - See also specific Synergy Types*
- SynergyWebProxy class 12-8 to 12-16
- SynergyWebProxy.xcall() method. *See* xcall() method
- synHandle class (Java) 12-44 to 12-45
- synImpDec class (Java) 12-46 to 12-47
- synnetcfg.exe 17-5
- synNull class (Java) 12-48
- SynProxyBadPropsException 11-3, 12-17
- SynProxyBadTypeException 11-3, 12-49
- SynProxyException 12-18
- SynProxyIncompatibleServerException 11-3, 12-19
- SynProxyInvalidStructException 11-3, 12-50
- SynProxyMixedTypeArrayException 11-3, 12-51
- SynProxyNetException 11-4, 12-3
- synrc file
 - comments in 2-48
 - setting XFPL_INIPATH in 3-20
 - setting XFPL_SMCPATH in 2-48
- synrc.com, setting XFPL_INIPATH in 3-21
- SYNSSL_RAND environment variable 3-24
- SYNSSLIB environment variable 3-25
- synStruct class (Java) 12-52

Index

T

syntlbcfg.exe 14-2

System.Collections.ArrayList class 1-16

System.Data.DataTable class 17-18

System.String class 1-13

T

technical support, obtaining xv

Telnet, using to debug remotely 3-41 to 3-44

test skeletons, generating 1-20 to 1-29

testing

 Synergy server code 1-20 to 1-29

 xfNetLink COM 15-11 to 15-12

 xfNetLink Java 11-9 to 11-10

 xfNetLink .NET 18-9 to 18-10

 xfNetLink Synergy 5-6 to 5-7

 xfServerPlus 3-15 to 3-16

time-outs

 in debug mode 6-6, 11-12, 15-13, 18-11

 KEEPALIVE 1-31

 recorded in xfServerPlus log 3-33

 using %RX_CONTINUE function 7-2

 xfNetLink COM 14-4, 14-12, 14-19

 xfNetLink Java 10-8 to 10-10, 10-19, 10-29

 xfNetLink .NET 17-7, 17-24, 17-30

 xfNetLink Synergy 5-3 to 5-5, 7-15

 xfServerPlus 1-31

TLB. *See* type libraries

Toolkit routines. *See* UI Toolkit routines

tools.def file 1-6

translating data. *See* data type mapping

troubleshooting. *See* debugging; errors; testing

type coercion

 defining in MDU 2-27, 2-33

 defining with attribute 2-10, 2-14

 xfNetLink COM B-10 to B-11

 xfNetLink .NET B-12 to B-14

See also data type mapping

type libraries 13-5 to 13-16

 creating from command line 13-11 to 13-15

 creating in Workbench 13-5 to 13-8

 excluding structure fields from 1-10

 registering

 from command line 13-16

 from Synergy TLB Configuration utility 14-2

 from Workbench 13-8

 unregistering

 from command line 13-16

 from Workbench 13-9

 using in an application 14-6 to 14-13

 viewing in Workbench 13-9

type mapping. *See* data type mapping; type coercion

type property 2-13

U

U_START subroutine 1-6

UI Toolkit routines 1-5 to 1-6

unregistering a type library

 from command line 13-16

 from Workbench 13-9

usePool() method (Java) 10-39

user interface elements in Synergy server code 1-5

utility methods

 xfNetLink COM 14-27 to 14-28

 xfNetLink Java 10-35 to 10-39

 xfNetLink .NET 17-38 to 17-41

V

^VAL parameters

 defining in MDU 2-34

 passing to %RXSUBR 7-24

vcvarsall.bat file 16-10, 16-20

verifying repository structure sizes in SMC 2-42

Visual Studio

 referencing Synergy assembly in project 17-9

 referencing xfnlnet.dll 17-9

 supported versions 16-1

 using application configuration file 17-5

 vcvarsall.bat file 16-10, 16-20

 vsvars32.bat file 16-10, 16-20

VMS. *See* OpenVMS

vsvars32.bat file 16-10, 16-20

W

web.config file 17-4

Windows application event log

 server-side logging 3-18, 3-29, 3-30, 6-3

 xfNetLink COM logging 15-1, 15-9, 18-1

 xfNetlink .NET logging 18-7

Workbench

 creating a COM component project 13-5 to 13-7

 creating a JAR file 9-4 to 9-8

- creating a Java component project 9-4 to 9-7
- creating a .NET component project 16-6 to 16-9
- creating a type library 13-5 to 13-8
- creating an assembly 16-5 to 16-11
- generating test skeletons 1-21 to 1-23
- registering a type library 13-8
- running an SMC/ELB comparison 2-58
- setting up .NET environment 16-10
- unregistering a type library 13-9
- viewing a type library 13-9

X

- xcall() method (Java)
 - method definition 12-13
 - using to call Synergy routines 12-15
- xercesImpl.jar file 9-3
- xf_DebugOutput setting 10-8, 11-7
- xf_DebugSessionConnectTimeout setting 10-9
- xf_LocalMaxPort setting 10-7
- xf_LocalMinPort setting 10-7
- xf_LogFile setting 10-8, 11-7
- XF_REMOTE_HOST environment variable 5-3
- XF_REMOTE_PORT environment variable 5-3
- xf_RemoteHostName setting 10-6, 12-17
- xf_RemotePort setting 10-6, 12-17
- XF_RMT_DBG_TIMEOUT environment variable 5-5
- XF_RMT_TIMEOUT environment variable 5-5
- XF_RMTCONN_TIMEOUT environment variable 5-5
- xf_SessionConnectTimeout setting 10-9
- xf_SessionLingerTimeout setting 10-10
- xf_SessionRequestTimeout setting 10-9
- xfJCWException 11-4, 12-57
- xfLaunchFatalException 11-4, 12-20 to 12-21
- xfMethod attribute 2-8 to 2-12
- xfNetLink COM
 - configuring type libraries 14-2 to 14-3
 - creating type libraries 13-5 to 13-8, 13-11 to 13-15
 - data type mapping B-8 to B-9
 - deploying application 14-1
 - error handling 15-1 to 15-8
 - host name, specifying 14-5
 - host port, specifying 14-5
 - logging 15-9 to 15-10
 - making remote calls 14-6 to 14-13
 - method reference 14-27 to 14-28
 - overview of tasks 13-3 to 13-4
 - pooling 14-14 to 14-26
 - registering type libraries 13-8, 13-16, 14-2
 - running xfServerPlus session in debug mode 15-12 to 15-15
 - technical overview 13-1 to 13-2
 - testing 15-11 to 15-12
 - time-outs 14-4, 14-12, 14-19
 - type coercion B-10 to B-11
 - unregistering type libraries 13-9, 13-16
 - See also* COM pooling; type libraries
- xfNetLink Java
 - ArrayList
 - using to pass binary data 10-17 to 10-18, 12-37 to 12-38
 - using to return structure collection 1-14 to 1-16
 - classes 12-1 to 12-58
 - code sample D-1 to D-5
 - data type mapping 12-43, B-1 to B-7
 - deploying application 10-1 to 10-2
 - exception handling 11-1 to 11-6
 - host name, specifying 10-6, 10-13
 - host port, specifying 10-6, 10-13
 - importing required packages 10-11, 10-22
 - logging 11-7 to 11-9
 - making remote calls 10-11 to 10-15
 - method reference 10-35 to 10-39
 - overview of tasks 9-1 to 9-3
 - pooling 10-20 to 10-34
 - properties file. *See* xfNetLink Java properties file
 - running xfServerPlus session in debug mode 11-11 to 11-15
 - setting classpath 9-3
 - technical overview 8-1 to 8-6
 - testing 11-9 to 11-10
 - time-outs 10-8 to 10-10, 10-19, 10-29
 - See also* JAR files; Java class wrappers; Java pooling
- xfNetLink Java properties file 10-2 to 10-10
 - comments in 10-4
 - compared with “set” methods 10-5 to 10-6
 - connection pooling and 10-27
 - errors in 12-17
 - naming 10-4, 10-27
 - placement of 10-4
 - settings
 - xf_DebugOutput 10-8
 - xf_DebugSessionConnectTimeout 10-9

Index

X

- xf_LocalMaxPort 10-7
- xf_LocalMinPort 10-7
- xf_LogFile 10-8
- xf_RemoteHostName 10-6
- xf_RemotePort 10-6
- xf_SessionConnectTimeout 10-9
- xf_SessionLingerTimeout 10-10
- xf_SessionRequestTimeout 10-9
- specifying for pooling 10-27
- used by SynergyWebProxy
 - constructors 12-10 to 12-13
- xfNetLink .NET
 - ArrayList, using to return structure
 - collection 1-14 to 1-16
 - configuration file 17-3 to 17-8
 - data type mapping B-12 to B-14
 - deploying application 17-1 to 17-2
 - error handling 18-1 to 18-6
 - Framework version in Workbench 16-10
 - host name, specifying 17-7
 - host port, specifying 17-8
 - logging 18-7
 - making remote calls 17-9 to 17-25
 - method reference 17-38 to 17-41
 - minimum version requirements 16-1
 - overview of tasks 16-3 to 16-5
 - pooling 17-26 to 17-37
 - running xfServerPlus session in debug
 - mode 18-10 to 18-13
 - System.Collections.ArrayList, passing 1-16
 - System.String, passing 1-13
 - technical overview 16-1 to 16-3
 - testing 18-9 to 18-10
 - time-outs 17-7, 17-24, 17-30
 - type coercion B-12 to B-14
 - See also* assemblies; C# classes; .NET pooling
- xfNetLink Synergy
 - API 7-1 to 7-28
 - code sample C-1 to C-5
 - configuring 5-2 to 5-6
 - error handling 6-3 to 6-4
 - logging 6-9 to 6-10
 - making remote calls 6-1 to 6-3
 - overview of tasks 5-1 to 5-2
 - running xfServerPlus session in debug
 - mode 6-5 to 6-9
 - technical overview 4-1 to 4-3
 - testing 5-6 to 5-7
 - time-outs 5-3 to 5-5, 7-15
- xfNetLnk.ini file. *See* xfNetLink Java properties file
- xfnljav.jar file 9-3
- xfnLJTest.class 11-9
- xfnlnet.dll 17-1, 17-2
 - referencing in Visual Studio project 17-9
- XFNLS_LOGFILE environment variable 5-6
- xfnlstst program 5-6
- xfParameter attribute 2-12 to 2-16
- xfpl_api library 1-30
- XFPL_BASECHAN setting 1-7
- XFPL_COMPRESS setting 3-22
- XFPL_DBR environment variable. *See Environment Variables and System Options*
- XFPL_DEBUG setting 3-36 to 3-38
- XFPL_FUNC_INFO setting 3-34 to 3-36
- XFPL_INIPATH environment variable 3-19 to 3-21
- XFPL_LOG setting 3-30
- XFPL_LOG subroutine 1-33
- XFPL_LOGFILE setting 3-30
- XFPL_LOGICAL setting 1-4
- XFPL_REGCLEANUP subroutine 1-34 to 1-36
- XFPL_SESS_INFO setting 3-31 to 3-34
- XFPL_SINGLELOGFILE setting 3-31
- XFPL_SMCPATH environment variable
 - setting for MDU 2-45
 - setting for xfServerPlus 2-46 to 2-49
- xfpl_tst.elb library 3-15, 5-6, 11-9
- xfpl.dbr 6-6, 11-13, 15-13, 18-11
- xfpl.ini file 3-18 to 3-21
 - default location 3-18
 - defining logicals in 1-4
 - errors in 3-18
 - settings
 - XFPL_BASECHAN 1-7
 - XFPL_COMPRESS 3-22
 - XFPL_DEBUG 3-36 to 3-38
 - XFPL_FUNC_INFO 3-34 to 3-36
 - XFPL_LOG 3-30
 - XFPL_LOGFILE 3-30
 - XFPL_LOGICAL 1-4
 - XFPL_SESS_INFO 3-31 to 3-34
 - XFPL_SINGLELOGFILE 3-31

- specifying location with
 - XFPL_INIPATH 3-19 to 3-21
- updating free pool after modifying (OpenVMS) 3-14
- using multiple 3-18
- xfPoolException 11-5, 12-58
- xfPool.properties file. *See* pooling properties file
- xfRemoteExecException 11-5, 12-22 to 12-23
- xfServer
 - using to access data remotely from
 - xfServerPlus 3-45 to 3-49
 - using to access remote SMC files 2-46, 2-50
- xfServerNackException 11-6, 12-4 to 12-5
- xfServerPlus
 - acting as client to xfServer 3-45 to 3-49
 - API 1-30 to 1-36
 - cleanup method 1-34 to 1-36
 - creating multiple sessions 3-7, 3-10, 3-13
 - data compression 3-22
 - debugging. *See* debug mode, running xfServerPlus in
 - default port 3-3
 - deploying application 3-44
 - ELB restrictions 1-3
 - encryption 3-23 to 3-28
 - error messages 3-16, 3-39 to 3-40
 - host name. *See* host name, defining on client
 - license requirements xiv
 - logging. *See* xfServerPlus log
 - overview of tasks 3-1 to 3-2
 - port. *See* host port number, defining on client; port number, xfServerPlus
 - protocol version 12-19
 - remote data access 3-45 to 3-49
 - running 3-3 to 3-14
 - running in debug mode. *See* debug mode, running xfServerPlus in
 - SET_XFPL_TIMEOUT subroutine 1-31
 - shutting down unexpectedly 1-34 to 1-36
 - starting
 - OpenVMS 3-12 to 3-14
 - UNIX 3-9 to 3-10
 - Windows 3-3 to 3-7
 - status 3-13, 3-16
 - stopping
 - OpenVMS 3-14
 - UNIX 3-11
 - Windows 3-8
 - testing 3-15 to 3-16
 - time-out, setting 1-31
 - XFPL_LOG subroutine 1-33
 - XFPL_REGCLEANUP subroutine 1-34 to 1-36
- xfServerPlus log 3-29 to 3-38
 - default name and location 3-30
 - deleting data from 3-30
 - error messages in 3-39 to 3-40
 - log file not created 3-31
 - making application-defined entries in 1-33
 - naming 3-30
 - setting options for 3-30 to 3-38
 - single vs. multiple files 3-31
 - specifying location for 3-30
 - turning on and off 3-30
 - writing to from COM client 14-12
 - writing to from Java client 10-20
 - writing to from .NET client 17-24
- XFPL_DEBUG 3-36 to 3-38
- XFPL_FUNC_INFO 3-34 to 3-36
- XFPL_LOG 3-30
- XFPL_LOGFILE 3-30
- XFPL_SESS_INFO 3-31 to 3-34
- XFPL_SINGLELOGFILE 3-31
- xfServerPlusUnavailableException 11-6, 12-6
- xfspl 3-6, 3-8
- xfsplst program 3-15 to 3-16
- xfTestchm program 15-11 to 15-12
- xfTestnet program 18-9 to 18-10
- XML file
 - for .NET API documentation 16-27 to 16-29
 - using to import/export SMC methods 2-39 to 2-41
 - See also* genxml utility
- xml-apis.jar 9-3

