

***xf*NetLink & *xf*ServerPlus User's Guide**

Version 10.1

Printed: June 2013

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, 2001–2013 by Synergex

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

Java is a trademark or registered trademark of Oracle and/or its affiliates in the U.S. and other countries.

Windows 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-01-10.1_02

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	xi
Other Resources.....	xii
Product Support Information	xii
Synergex Professional Services Group	xiii
Comments and Suggestions.....	xiii

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
Passing Enumerations	1-12
Handling Variable-Length and Large Data.....	1-13
Passing a Single Parameter as a Memory Handle.....	1-13
Passing a System.String Parameter	1-14
Returning a Collection of Structures	1-15
Passing a System.Collections.ArrayList Parameter	1-16
Passing Arrays Larger Than 64K.....	1-18
Passing Binary Data	1-18
Handling Errors.....	1-19
Testing Your Synergy Code.....	1-21
Generating Test Skeletons with Workbench	1-22
Generating Test Skeletons from the Command Line	1-24
The gensyn Utility.....	1-25
Using the Test Skeletons	1-26
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	2-13
Attribute Examples	2-16
Documentation Comments.....	2-20
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.....	2-37
Importing and Exporting Methods	2-38
Verifying Repository Structure Sizes and Enumerations	2-41
Defining Multiple Synergy Method Catalogs.....	2-42
Creating New SMC Files.....	2-42
Specifying Which SMC to Update.....	2-44
Setting the XFPL_SMCPATH Environment Variable for xfServerPlus.....	2-44
The Method Definition Utility.....	2-48
The SMC/ELB Comparison Utility.....	2-53
Windows and UNIX	2-53
OpenVMS.....	2-55
Running an SMC/ELB Comparison from Workbench	2-56

3. Configuring and Running xfServerPlus

The Big Picture	3-1
Running xfServerPlus	3-2
Running xfServerPlus on Windows	3-2
Running xfServerPlus on UNIX	3-8
Running xfServerPlus on OpenVMS	3-11
Testing xfServerPlus	3-14
xfServerPlus Status Codes	3-15
Using the xfpl.ini File	3-17
Using an Alternate xfpl.ini File.....	3-17
Setting the XFPL_INIPATH Environment Variable	3-18
Configuring Compression	3-21
Using Encryption	3-22
Setting up the xfServerPlus Machine for Encryption	3-23
Setting up the xfNetLink Synergy Machine for Encryption	3-25

Setting up the <i>xfNetLink</i> Java Machine for Encryption	3-25
Setting up the <i>xfNetLink</i> .NET Machine for Encryption	3-28
Specifying the Data to Encrypt for Slave Encryption.....	3-29
Using Server-Side Logging	3-30
Setting Options for the <i>xfServerPlus</i> Log	3-31
Error Messages in the <i>xfServerPlus</i> Log.....	3-41
Debugging Your Remote Synergy Routines	3-43
Debugging Remote Synergy Routines via Telnet	3-45
Deploying Your Distributed Application	3-47
Deploying the Server.....	3-47
Deploying the Client	3-48
Configuring <i>xfServerPlus</i> for Remote Data Access.....	3-49
Remote Data Access When <i>xfServerPlus</i> Is on Windows.....	3-49
Remote Data Access When <i>xfServerPlus</i> Is on UNIX	3-51

Part II: *xfNetLink* Synergy Edition

4. Configuring & Testing *xfNetLink* Synergy

System Overview	4-1
The Big Picture.....	4-2
Configuring <i>xfNetLink</i> Synergy	4-4
Specifying the Host Name and Port Number.....	4-4
Specifying Time-out Values.....	4-5
Specifying Debug Options	4-7
Testing <i>xfNetLink</i> Synergy	4-8

5. Calling Synergy Routines Remotely from Synergy

Making Remote Calls	5-1
Making Remote Calls with %RXSUBR	5-1
Making Remote Calls Using a Routine Call Block.....	5-3
Handling Errors.....	5-3
Troubleshooting Techniques	5-4
Running an <i>xfServerPlus</i> Session in Debug Mode	5-4
Viewing Packets.....	5-8

6. *xfNetLink* Synergy API

%RX_CONTINUE	6-2
%RX_DEBUG_INIT	6-4
%RX_DEBUG_START	6-5
RX_GET_ERRINFO	6-7
RX_GET_HALTINFO.....	6-9
%RX_RMT_ENDIAN	6-11
%RX_RMT_INTSIZE	6-12

Contents

%RX_RMT_OS	6-13
%RX_RMT_SYSINFO	6-14
%RX_RMT_TIMEOUT	6-15
RX_SETRMTFNC	6-16
RX_SHUTDOWN_REMOTE	6-17
%RX_START_REMOTE	6-18
%RXSUBR	6-23

Part III: *xf*NetLink Java Edition

7. Creating Java Class Wrappers

System Requirements	7-1
System Overview	7-1
The Big Picture	7-3
Setting the Classpath	7-5
Creating a Java JAR File in Workbench	7-5
Creating a Synergy/DE Java Component Project	7-6
Generating Java Class Wrappers	7-9
Building the JAR File	7-10
Creating a Java JAR File from the Command Line	7-11
The genxml Utility	7-11
The genjava Utility	7-14
Building the JAR File	7-16
Understanding the Generated Classes	7-18
Procedural Classes	7-18
Structure Classes	7-19
Enumeration Classes	7-19
Editing the Java Source Files	7-20
Generating Javadoc	7-20

8. Calling Synergy Routines from Java

Deploying Your Distributed Application	8-1
Deploying the Server	8-1
Deploying the Client	8-1
Configuring the <i>xf</i> NetLink Java Properties File	8-3
Creating and Naming a Properties File	8-4
Using a Properties File vs. Using the “set” Methods	8-5
Specifying the Host Name and Port Number	8-6
Specifying Logging Options	8-7
Specifying Time-out Values	8-7
Specifying Encryption Options	8-9

Using Your JAR File	8-10
Using Structures	8-14
Using Enumerations	8-16
Passing Binary Data	8-17
Setting a Call Time-Out	8-19
Writing to the <i>xfServerPlus</i> Log	8-20
Understanding Java Pooling	8-21
Implementing Pooling	8-22
Using Your JAR File with Connection Pooling	8-22
Setting Up a Pooling Properties File	8-24
Pool Maintenance	8-31
Using the Pooling Support Methods	8-32
Method Reference	8-35
Class Reference	8-40
Synergex.util.SWPCConnect	8-40
Synergex.util.SWPMManager	8-42
Synergex.util.xfJCWException	8-45
Synergex.util.xfPoolException	8-46
9. Error Handling and Troubleshooting in <i>xfNetLink</i> Java	
Handling Errors	9-1
Troubleshooting Techniques	9-7
Using Client-Side Logging	9-7
Testing <i>xfNetLink</i> Java	9-9
Running an <i>xfServerPlus</i> Session in Debug Mode	9-10

Part IV: *xfNetLink* .NET Edition

10. Creating Synergy .NET Assemblies

System Requirements	10-1
System Overview	10-2
The Big Picture	10-3
Creating an Assembly in Workbench	10-6
Creating a Synergy/DE .NET Component Project	10-6
Controlling the .NET Environment	10-11
Generating C# Classes	10-11
Building the Assembly	10-12
Creating an Assembly from the Command Line	10-14
The genxml Utility	10-14
The gensc Utility	10-17
Building the Assembly	10-21

Contents

Understanding the Generated Classes.....	10-23
Procedural Classes	10-23
Structure Classes.....	10-24
DataTable Classes	10-24
Enumerations	10-25
Custom Attributes	10-25
Using Your Own Key File.....	10-25
Editing the Generated Files.....	10-26
Editing Information in AssemblyInfo.cs	10-26
Generating API Documentation.....	10-27
Adding Documentation Comments	10-27
Generating an XML File	10-28
Creating the API Documentation	10-29

11. Calling Synergy Routines from .NET

Deploying Your Distributed Application.....	11-1
Deploying the Server	11-1
Deploying the Client (for Development)	11-1
Deploying the Client (at a Customer Site)	11-2
Using an Application Configuration File	11-3
Creating and Editing Configuration Files	11-4
Using Your Synergy .NET Assembly.....	11-9
Using Multiple Copies of the Same Class.....	11-12
Using Structures	11-13
Using DataTables.....	11-17
Using Enumerations	11-21
Passing Binary Data	11-22
Setting a Call Time-Out	11-23
Writing to the <i>xfServerPlus</i> Log	11-24
Understanding .NET Pooling	11-25
Object Pooling	11-25
Reusing Objects	11-25
Tips for Creating Poolable Objects.....	11-26
Implementing Pooling	11-27
Implementation Overview	11-27
Setting Up the Client Machine	11-27
Changing the Pool Configuration	11-31
Using the Pooling Support Methods.....	11-31
Writing Code That Uses Pooled Objects	11-34
Method Reference	11-37
Procedural Methods.....	11-37
Structure Methods	11-39
DataTable Methods.....	11-39

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

Handling Errors.....	12-1
Troubleshooting Techniques	12-7
Using Client-Side Logging	12-7
Testing <i>xfNetLink</i> .NET	12-9
Running an <i>xfServerPlus</i> Session in Debug Mode	12-10

Appendices

A. Configuration Settings

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

B. Data Type Mapping

<i>xfNetLink</i> Java	B-1
<i>xfNetLink</i> .NET	B-6

C. *xfNetLink* Synergy Sample Code

Client Application (synclt.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[™], or .NET client. There are three *x/NetLink* clients from which you can access Synergy logic.

- ▶ *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.
- ▶ *x/NetLink* Java Edition works in conjunction with the 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 that references your Synergy routines. The user interface can be presented by a Java application, a Java applet, or JavaServer Pages[™].
- ▶ *x/NetLink* .NET Edition enables a .NET client to call Synergy routines residing on a remote server. Using the component generation tools, the .NET Framework, and Microsoft® Visual Studio®, you can create an assembly that references Synergy routines. *x/NetLink* .NET is supported only on Windows platforms; 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
- ▶ *x/NetLink*. You'll need one of the following:
 - ▶ *x/NetLink* Synergy: Synergy routines for connecting a Synergy client to a remote Synergy server.
 - ▶ *x/NetLink* Java: tools to generate Java class wrappers and create a JAR file for connecting a Java client to a remote Synergy server.
 - ▶ *x/NetLink* .NET: tools to generate .NET assemblies for connecting a .NET client application to a remote Synergy server.

- ▶ *xfServerPlus*. You must have at least one *xfServerPlus* license for each machine you want to use a server. The *xfServerPlus* installation also includes
 - ▶ The Method Definition Utility (MDU) for defining Synergy functions and subroutines that you wish to call remotely.
 - ▶ *xfServerPlus* Application Program Interface (API) for setting remote execution options.



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

Other Resources

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

- ▶ Release notes for *xfNetLink* Java, *xfNetLink* .NET, and *xfServerPlus*.
- ▶ Synergy DBL 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.
- ▶ Tutorials for the *xfNetLink* products. Check the [Synergex website](#) to see which tutorials are currently available.

Product Support Information

If you cannot find the information you need in this manual or in the resources listed above, you can reach the Synergy/DE Developer Support department at the following numbers:

800.366.3472 (in the U.S. and Canada)
916.635.7300 (in all other locations)

To learn about your Developer Support options, contact your Synergy/DE account manager at one of the above numbers.

Before you contact us, collect as much of following information as possible:

- ▶ 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.
- ▶ Statement at which the error occurred.
- ▶ Exceptions thrown by *xfNetLink* Java or *xfNetLink* .NET.
- ▶ Contents of the *xfServerPlus* log or the application event log (Windows) or syslog (UNIX).
- ▶ Contents of the client-side log.
- ▶ 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 can be reproduced 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 the U.S. and Canada) 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 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 DBL system-supplied 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 code, see the Synergex white paper, *Modularizing Your Synergy Code: The First Step to Distributed Computing*, available on the [Synergex website](#).

Attributing Your Code

Attributes (introduced in version 9.3) are a feature of Synergy DBL 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 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 *xfServerPlus* can find the routine. The ELBs are opened and closed by *xfServerPlus*.

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 *xfServerPlus*. (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 Traditional Synergy Applications” chapter in *Synergy 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 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 Traditional Synergy Applications](#)” chapter in *Synergy 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 *xfServerPlus* knows how to resolve them.

Defining logicals on Windows and UNIX

On Windows and UNIX, define 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 3-17.)

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

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=LN\$RSDMS\$MGR_port /USER LOGICAL_NAME value
where *port* is the port number on which *xfServerPlus* is running.

For additional information about defining logicals on OpenVMS, see [“Defining logical names for xfServer processes”](#) 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 because 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 Traditional Synergy Applications” chapter of *Synergy Tools* for more information about **db**s, including specifics on limited and unavailable functionality.)

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_PUTTEXT**). 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**.

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 and Functions” chapter of the *Synergy DBL 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.

Preparing Your Synergy Server Code

Removing User Interface Elements

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

For example, when opening the SMC files (**cdt.ism** and **cmpdt.ism**), *xfServerPlus* 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 *xfServerPlus*, 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
```

xfServerPlus will attempt to open the SMC files on channels 150 and 151. If 150 is currently in use, *xfServerPlus* will try 151, then 152, and so on until it finds an open channel. Should *xfServerPlus* reach channel 255 without finding an open channel, it will go to $\text{XFPL_BASECHAN} - 1$ (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 *xfNetLink* Java or *xfNetLink* .NET. *xfNetLink* supports passing structures of primitive, date, and time types; structures with embedded structures (that is, groups or struct data types) or embedded arrays; arrays of structures; and enumeration types within structures. User-defined fields are generally treated as strings, except for some specific datetime formats. See [“Appendix B: Data Type Mapping”](#) for complete information on supported data types and for details on how repository field data types are mapped from Synergy to *xfNetLink*.

Regardless of whether you create a JAR file or an assembly, 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 *xfNetLink* .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 10-6 or [“The gens Utility”](#) on page 10-17.)

By default, the repository field name becomes the property name. To specify an alternate name, enter it in the “Alt name” field (on the Display tab) 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 *xfNetLink*, 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 DBL 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 *xfNetLinks*, refer to the following:

- Java, see [page 8-14](#)
- .NET, see [page 11-13](#)

If your Java or .NET client passes structures as parameters, and you want to also use an *xfNetLink* 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
```

However, if you have this structure in your repository

```
customer
  name           ,a51
  firstname       ,a20 @ name
```

Preparing Your Synergy Server Code

Passing Structures as Parameters

```
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. (See [“Basic field information”](#) in the “Working with Fields” chapter of the *Repository User’s Guide* for information on this flag.) 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 a field 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 [“The Field Definitions list”](#) in the “Working with Fields” 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 “[Working with Structures](#)” and “[Working with Fields](#)” chapters of the *Repository User’s Guide* for instructions.
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 MDU’s Verify Catalog utility to update the structure sizes. See “[Verifying Repository Structure Sizes and Enumerations](#)” on page 2-41 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 *xfNetLink* client you are using:
 - Java, see [chapter 7, “Creating Java Class Wrappers”](#)
 - .NET, see [chapter 10, “Creating Synergy .NET Assemblies”](#)

Passing Enumerations

You can pass enumerations defined in the repository as parameters or return values, as well as include them as fields in a structure passed as a parameter. This feature is supported on *xfNetLink* Java and *xfNetLink* .NET. In Java, enumerations are included in your JAR file as enum type classes; in .NET, they are included in the assembly as enumeration types, or enums.

Regardless of whether you are creating a JAR file or an assembly, the repository enumeration becomes a class (named with the enumeration name), and each member of the enumeration is accessible from your client application.

If you assign numerical values to the members in the repository, they are used for the integer equivalents assigned to the values in the generated class; else, values are assigned automatically starting with 0 and incrementing by 1.

When you create a new instance of an enumeration in your client code, it has a default value of the enumerator that has been assigned 0, if you do not explicitly assign a value. Consequently, when defining your enumeration in the repository, you should specify as the first member the value you would like to be the default.

In your Synergy code, you must `.INCLUDE` the enumeration.

For more information on using enumerations with the individual *xfNetLinks*, refer to the following:

- ▶ Java, see [page 8-16](#)
- ▶ .NET, see [page 11-21](#)

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.
- ▶ (Java and .NET clients 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 1-14.
- ▶ (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 1-15.
- ▶ (Java and .NET clients 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 *xfNetLink* clients can handle arrays larger than 64K, but only the Synergy client requires a special technique to do so.) See [“Passing Arrays Larger Than 64K”](#) on page 1-18.

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

Preparing Your Synergy Server Code

Handling Variable-Length and Large Data

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 client, the parameter is handled as a `String` (or `StringBuffer`); on a .NET client, it is handled as a string. See the data type mapping tables in “[Appendix B: Data Type Mapping](#)” for details.
- ▶ 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 DBL 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 Java or .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 DBL see “[Data Types](#)” in the “Defining Data” chapter of the *Synergy DBL Language Reference Manual* and [System.String](#) in the “System-Supplied Classes” chapter of that same manual.
- ▶ 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 Java or .NET documentation for more information.

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.

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 11-17](#) for more information.

Preparing Your Synergy Server Code

Handling Variable-Length and Large Data

- ▶ On your Java 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. If you set the “Generate classes as version” option in Workbench to 1.5 (or run **genjava** with the **-c 1.5** option), the ArrayList will be generic, so you must instantiate an ArrayList of structures. When the structure collection parameter is returned, use the `ArrayList.size()` method 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 documentation for more information on ArrayLists.
- ▶ On your .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. If you run **gens** with the **-w** option, you will need to instantiate a `List<T>` instead of an ArrayList. When the structure collection parameter is returned, use the `ArrayList.Count` (or `List<T>.Count`) property to get the size. Then you can enumerate through the ArrayList to access the structures or access them by position (index). Refer to your .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 client with your server-side code, note the following:

- ▶ 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 *xfServerPlus* using the `RCB_XXX` routines. (See [“Passing a Single Parameter as a Memory Handle” on page 1-13.](#))
- ▶ Do not 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 structure collection in the SMC.

Passing a `System.Collections.ArrayList` Parameter

When using a Java or .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 1-15.](#) 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 Classes” chapter of the *Synergy DBL 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](#).



In *xfNetLink .NET*, 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 11-17](#) for more information.

- ▶ For Java, you must set the “Generate classes as version” option in Workbench to 1.5 (or run **genjava** with the -c 1.5 option). In your client code, instantiate an ArrayList (generic, of the specified type). Mixed type ArrayLists are not supported.
- ▶ For .NET, in your client code, instantiate an ArrayList or, if you ran **gens** with the -w option, a List<T>. Mixed type ArrayLists are not supported. Refer to your .NET documentation for more information on Lists and 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 DBL is 256 MB. This feature is supported on all clients.

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).
- ▶ For a Java or .NET client, you do not need to do anything special on the client side to pass large arrays.
- ▶ For 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 DBL Language Reference Manual*.

Passing Binary Data

You can pass binary data, such as JPEG files, when using a Java or .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 10-19](#).

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

- ▶ Java, see [page 8-17](#)
- ▶ .NET, see [page 11-22](#)

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 client with your server-side code, note the following:

- ▶ 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 1-13](#).)
- ▶ 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.

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.

Preparing Your Synergy Server Code

Handling Errors

To help you deal with the complexities of troubleshooting in a distributed environment, 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 3-30](#).) This enables the system administrator to track problems that have occurred even if they are not reported by a user. You can also activate client-side logging. For information on handling errors for the individual *xfNetLink* clients, refer to the following:

- ▶ Synergy, see [page 5-3](#)
- ▶ Java, see [page 9-1](#)
- ▶ .NET, see [page 12-1](#)

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 *xfNetLink* Synergy client before adding a Java 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 *xfNetLink* 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 *xfNetLink* Java or *xfNetLink* .NET. If you are using *xfNetLink* 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



In version 9.3 and later, the Generate Synergy Test Skeletons menu entry is not included in new projects. 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 1-24](#)). 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 or .NET Component project.

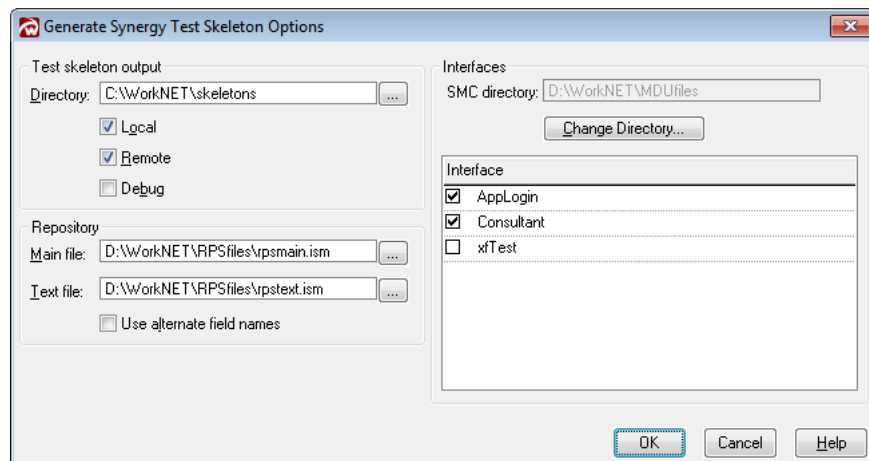


Figure 1-1. The Generate Synergy Test Skeletons Options dialog box.

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.

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 3-43](#) for information about the Telnet method.)

Repository main file. 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. 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 XFPL_SMCPATH; if it is not set, the default is DBLDIR. To change the SMC directory, click the Change Directory button to display the Browse for SMC

Preparing Your Synergy Server Code

Testing Your Synergy Code

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 1-26](#) 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 an 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 7-11](#) or [page 10-14](#) 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 an 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 1-25](#) for the complete syntax. For example, on Windows:

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

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 1-26](#) 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 `dbf 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 *x/NetLink* Synergy and *x/ServerPlus* 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 *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 3-43](#) for information about the Telnet method.

Preparing Your Synergy Server Code

Testing Your Synergy Code

-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.

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 *xfNetLink* Synergy API. The API, along with information on setting up your system, making remote calls, and troubleshooting *xfNetLink* 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 *xfServerPlus* 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 *xfServerPlus* 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.

Preparing Your Synergy Server Code

Testing Your Synergy Code

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 DBL (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 6-20](#) for status codes returned during session start-up or the table on [page 6-26](#) 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 4-8](#) and “Testing *xfServerPlus*” on [page 3-14](#).

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 5-4](#) 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 5-7](#).

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 5-4](#).

If you encounter errors while running the debug program, check the table on [page 6-20](#) for status codes returned during session start-up or the table on [page 6-26](#) 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 a license.

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 *xfNetLink* clients:

- ▶ Synergy, see [“Specifying Time-out Values” on page 4-5](#)
- ▶ Java, see [“Specifying Time-out Values” on page 8-7](#)
- ▶ .NET, see [“Setting a Call Time-Out” on page 11-23](#)

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 and supplying a logfile name. See [“Using Server-Side Logging” on page 3-30](#) 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 2-38](#).

XFPL_LOG must be included in the SMC if you are using the `setUserString()` methods in *xfNetLink* Java 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 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 2-38](#).

► 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 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 Java JAR file or .NET assembly, 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 JAR file or assembly 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 or .NET client. (For general information about pooling, see [“Understanding Java Pooling” on page 8-21](#) or [“Understanding .NET Pooling” on page 11-25](#).)

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, nor do you need to 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 8-32](#) and [“Cleanup method” on page 8-34](#)
 - ▶ .NET, see [“Using the Pooling Support Methods” on page 11-31](#) and [“Cleanup method” on page 11-34](#)
3. Use the MDU to add your cleanup routine to the Synergy Method Catalog.
 - ▶ 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.
 - ▶ For .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 assembly.

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 2-42.) 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 routines.
- ensure that data from *xfNetLink* clients is translated into the correct Synergy DBL 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
- 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 also to offer some suggestions on how best to use these fields depending on which *x/NetLink* 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 *x/NetLink* Java and *x/NetLink* .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 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. *x/ServerPlus* 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 and *xfNetLink* .NET never directly reference the method ID in the client code. (However, Java connection pooling does require that you reference the method ID in the pooling properties file.) Users of *xfNetLink* Synergy 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 uses the method ID to invoke the Synergy routine, it does 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.

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

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 2-20](#).

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.db1 -out c:\work\Vendor.xml
```

See “[The Dbl2xml Utility](#)” in the “General Utilities” chapter of *Synergy 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** 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,

```
dbr dblDir: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 *xfServerPlus* 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 2-10 .	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 2-11 . 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
Parameter name	Defaults to the declared Synergy parameter name, but can be overridden with the name property. See page 2-13 .	xfParameter	name="xxx"

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

Item	Source/Comments	Attribute	Property
Parameter data type	<p>Obtained from the parameter definition for most data types. In some cases, must be specified with the type property. See page 2-13.</p> <p>To coerce the parameter to a non-default type on the client, use the cType property; see page 2-15. Parameters that are coerced to DateTime require the format property; see page 2-15. A DateTime parameter can be created as a nullable type on the client using the nullable property; see page 2-15.</p> <p>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.</p> <p>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.</p> <p>Group arguments defined in the data division are processed as a single field of the type and size specified. If no type is specified, it defaults to alpha. Group arguments included from the repository are processed as a single field of type alpha, with the size taken from the repository. See the note on page 2-17.</p>	xfParameter	type=SynType.xxx cType=xfType.xxx format=xfFormat.xxx nullable=true collectionType=xfCollectType.xxx structure="xxx" dataTable=true
Parameter size	<p>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 2-14.</p>	xfParameter	length=xxx precision=xxx
Parameter direction	<p>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 DBL Language Reference Manual</i>.</p>	N/A	N/A
Parameter required/optional	<p>Obtained from parameter modifiers (REQ, OPT) in the source code. If not supplied, defaults to required. (For Java and .NET clients, any parameter marked optional in the SMC is changed to required when the component is built.)</p>	N/A	N/A

xfMethod Attribute

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

The `xfMethod` attribute describes a subroutine or function (Synergy object-oriented 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.



If you are attributing your code for use with a Synergy/DE Interop project in Visual Studio, only the `interface` property of the `xfMethod` attribute is required. The `elb`, `id`, and `encrypt` properties of the `xfMethod` attribute are not used. Other properties may be necessary depending on your code. For more information about interop projects, see [“Converting xfServerPlus routines for native .NET access”](#) in the “Developing for the .NET Framework” chapter of *Getting Started with Synergy/DE*.

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 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 DBL, 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.

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

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 are 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 *xfServerPlus*, 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 Java and .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 2-18](#). (This example shows type coercion for a parameter, but the principal is the same for the return value.) See [“Appendix B: Data Type Mapping”](#) 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 DBL 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 DBL Language Reference Manual* for more information on these types.

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

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

Return value data type	Valid coerced types	
	<i>xfNetLink</i> Java	<i>xfNetLink</i> .NET
Integer (i)	byte short int (coerced to Integer) long Boolean	byte short int long sbyte ushort uint ulong Boolean

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. These values are case insensitive.

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

nullable=true

Indicates that a nullable DateTime or decimal return type is desired on the client side. This option is supported for .NET clients only. The nullable property is valid only when the cType property is DateTime or decimal. “False” is a valid value and is the same as not setting the property.

encrypt=true

Indicates that encryption is desired for this method. 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 3-22](#) 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 sensitive.)



Although the \$ character is valid in parameter names in Synergy DBL, it is not valid in parameter names in the SMC. If you use \$ in 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. Supported on Java and .NET clients only. See [example G on page 2-19](#).

When using this property, the Synergy parameter must be a memory handle (i4; do not use int). 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 1-13](#)
- ▶ [“Passing Binary Data” on page 1-18](#)

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

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.

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 [“Supported Parameter Data Types and Collection Types by Client” 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 Java and .NET clients only. See the table on [page 2-11](#) for the valid cType values for each data type. See [example D on page 2-18](#). See [“Appendix B: Data Type Mapping”](#) 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 DBL 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 DBL 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 or decimal parameter is desired on the client side. This option is supported for .NET clients only. The nullable property is valid only when the cType property is DateTime or decimal. “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 or a structure collection. This feature is 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

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

See [example H on page 2-19](#) for a structure collection, [example C on page 2-18](#) 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 2-19](#).



If you rename structures using a .INCLUDE directive 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 **db12xml** utility will map the name in the property to the original name and write the latter to the XML. See [example C on page 2-18](#).

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 Data Tables" on page 11-17](#) 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 DBL Language Reference Manual* for information on structfields.)



When an attribute follows a `.INCLUDE` directive, 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="Cust", 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
```



If you do this in your code

```
.subroutine mysub
    req in arg1    ,a10
    .include "fred" repository, req out group="arg2"
    req out arg3   ,d8
endparams
```

the parameter included from the repository will *not* be processed as a structure. Rather, it will be processed as a *single* alpha field of the size specified in the repository. If you want to preserve the fields in a group argument when you generate classes for the client side, you should `.INCLUDE` the structure globally and then define the parameter as a structfield, as shown above in [example B](#).

Defining Your Synergy Methods

Using Attributes to Define Synergy Methods

- 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
```

- 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 1-13](#) 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 an ArrayList on a Java client and a byte array on a .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 .NET client. In the Synergy code, the parameter is defined as a memory handle (see [page 1-15](#)). 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 *xfNetLink* Java or *xfNetLink* .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 *xfNetLink* Synergy. (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 7-20](#)
- ▶ .NET, see [“Generating API Documentation” on page 10-27](#)

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:

```
;;; <summary>This is a comment</summary>
```
- ▶ The text for each type of comment is limited to 6 lines of 50 characters each.
- ▶ The comment tags may be placed on the same line as the text or on separate lines.
- ▶ 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. Do not put a documentation comment on the same line as the *xfMethod* attribute; it will be ignored.

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. This comparison is *case sensitive*.

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>  
;;;<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 2-38) and to verify repository structure sizes (see [“Verifying Repository Structure Sizes and Enumerations”](#) on page 2-41).

On Windows, you can start the MDU from Workbench or—on 32-bit platforms—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 2-48.

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 2-41](#).)

1. Start the Method Definition Utility. See [page 2-48](#) for the command line syntax.

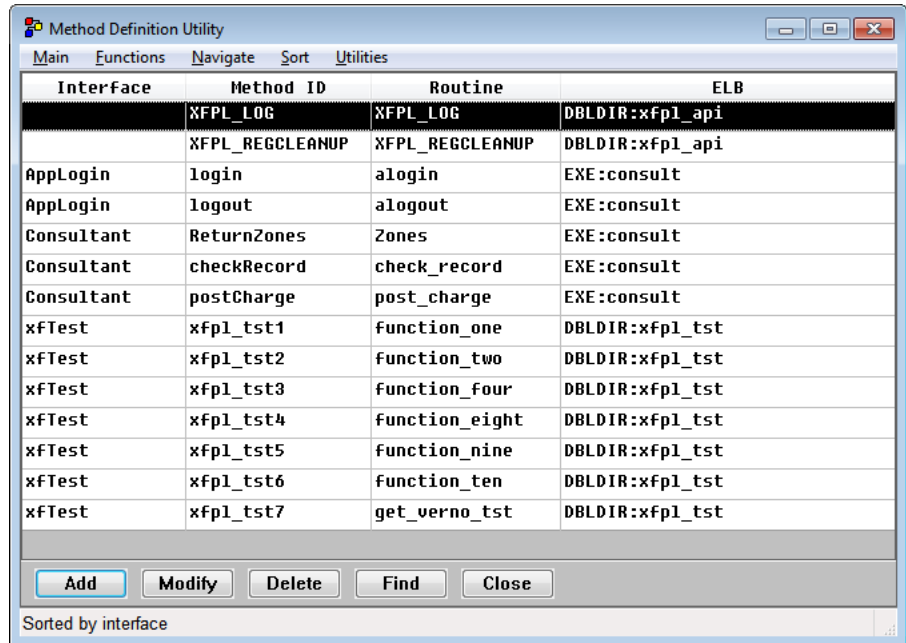
When the MDU opens, the methods list (see [figure 2-1](#)) displays the methods already in the catalog. The SMC is pre-loaded with several methods: XFPL_LOG, XFPL_REGCLEANUP, and a number of methods needed to run the *xfNetLink* and *xfServerPlus* test programs. (The methods for the test programs are 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 ID	Routine	ELB
	XFPL_LOG	XFPL_LOG	DBLDIR:xfpl_api
	XFPL_REGCLEANUP	XFPL_REGCLEANUP	DBLDIR:xfpl_api
AppLogin	login	alogin	EXE:consult
AppLogin	logout	alogout	EXE:consult
Consultant	ReturnZones	Zones	EXE:consult
Consultant	checkRecord	check_record	EXE:consult
Consultant	postCharge	post_charge	EXE:consult
xfTest	xfpl_tst1	function_one	DBLDIR:xfpl_tst
xfTest	xfpl_tst2	function_two	DBLDIR:xfpl_tst
xfTest	xfpl_tst3	function_four	DBLDIR:xfpl_tst
xfTest	xfpl_tst4	function_eight	DBLDIR:xfpl_tst
xfTest	xfpl_tst5	function_nine	DBLDIR:xfpl_tst
xfTest	xfpl_tst6	function_ten	DBLDIR:xfpl_tst
xfTest	xfpl_tst7	get_verno_tst	DBLDIR:xfpl_tst

Sorted by interface

Figure 2-1. Displaying the methods list.

- 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](#)).

- 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. The maximum length is 50 characters.

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

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

The screenshot shows a window titled "Method Definition" with a "Date last updated" label. It 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
- ☐ LINQ stored procedure

Synergy routine information:

- Routine name: alogin
- ELB name: EXE:consult
- Return type: Decimal (dropdown), Length: 4, Precision: (empty)
- Coerced type: Boolean (dropdown), Format: (empty)
- Return desc: Returns success or failure

At the bottom, there is a checkbox for "Enable encryption" which is checked. To the right, it shows "Maximum number of parameters: 3" and "Number of required parameters: 2". At the very bottom are buttons for "OK", "Cancel", "Method ID", and "Parameter".

Figure 2-2. Completing the Method Definition window.

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.

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 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 or assembly.

The interface name is also used to select methods when you generate test skeletons. (See [“Testing Your Synergy Code”](#) on page 1-21.) 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** or **gencs** 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., MYINTFACE, 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 7-20](#) or [“Generating API Documentation” on page 10-27](#) for more information on generating documentation.) If you are using a Synergy 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 **gencs**, 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.

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

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.

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

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 DBL data type of the returned data. For subroutines, select “No return value”. System.String and Enumerations are supported only on Java and .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 2-48](#).

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 blank.
- ▶ 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 Java and .NET clients only. On Java, a DateTime coerced type is mapped to the Calendar class and decimal is mapped to BigDecimal; nullable types are not supported on Java. Select “Default” to use default type mapping. See [“Appendix B: Data Type Mapping”](#) for more information on data type mapping and coercion.

Format. If the coerced type is DateTime or nullable DateTime, select the desired format.

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 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 3-22](#) 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 2-34](#) 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* (supported on Java and .NET clients only), 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 Java and .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 2-48](#).

Description. (optional) Enter a description of the parameter. 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 7-20](#) or [“Generating API Documentation” on page 10-27](#) for more information on generating documentation.) If you are using a Synergy client, 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 DBL 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 1-13](#) and [“Returning a Collection of Structures” on page 1-15](#).

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.

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

- ▶ For structures and enumerations, the Length field is completed automatically with information from the repository.
- ▶ For arrays and ArrayLists that are not made up of structure elements, enter the element length, not the total array size.

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	.NET
Alpha	✓	✓	✓
Decimal	✓	✓	✓
Implied-decimal	✓	✓	✓
Integer	✓	✓	✓
Structure		✓	✓
Handle	✓	✓	✓
Binary (handle)		✓	✓
System.String		✓	✓
Enumeration		✓	✓
Array	✓	✓	✓
ArrayList		✓	✓
Structure collection		✓	✓
DataTable			✓

- For decimal parameters coerced to DateTime or nullable DateTime data types, the length is determined by the selected DateTime format.
- For handle, binary (handle), and System.String parameters, this field is blank.

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 Java and .NET clients only. On Java, a DateTime coerced type is mapped to the Calendar class and decimal is mapped to BigDecimal; nullable types are not supported on Java. Select “Default” to use the default type mapping. See [“Appendix B: Data Type Mapping”](#) for more information on data type mapping and coercion.

Format. If the coerced type is DateTime or nullable DateTime, select the desired format.

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.

xfNetLink Synergy supports multi-dimensional arrays (up to 9 dimensions); the number of dimensions and elements per dimension are calculated from the data passed. *xfNetLink* Java and *xfNetLink* .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 1-18](#) for more information.

ArrayList. If the parameter is a Synergy System.Collections.ArrayList class, select this field. Supported on Java and .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 *xfServerPlus* 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 1-15](#) 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 11-17](#) for more information.

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 *xfServerPlus*. 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 *xfServerPlus* 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 *xfServerPlus* 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.

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

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 *x/NetLink Java* and *x/NetLink .NET*. When **genjava** or **gens** 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.

► 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.

Defining Your Synergy Methods

Using the MDU to Define Synergy Methods

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 *xfServer*, 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 2-40](#)).

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 2-43](#) 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.

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.

Defining Your Synergy Methods

Importing and Exporting Methods

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 2-41](#) 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 2-48](#) 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 2-48](#).

Defining Multiple Synergy Method Catalogs

By default, the Method Definition Utility and *xfServerPlus* 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 *xfServerPlus* 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 *xfServerPlus* 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.

► 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 2-38.)



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 2-40) 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:

```
dbf 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 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 & 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 2-44](#)).

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 2-37](#).



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 2-44](#).

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 Windows Control Panel, select Synergy Control Panel > 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.

Defining Your Synergy Methods

Defining Multiple Synergy Method Catalogs

► 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 Windows Control Panel, select Synergy Control Panel > 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` (or any environment variable) 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 for *xfServer* processes”](#) 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. See the examples starting on [page 2-51](#).

Syntax `dbm mdu [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 2-44](#)). 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 [example D on page 2-51](#).

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 2-37](#).

-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 x/Server, include the node name in *rps_main*. See [example E on page 2-52](#).

-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 x/Server, include the node name in *rps_text*. See [example E on page 2-52](#).

-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**. See [example G on page 2-52](#).

-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**. See [example H on page 2-52](#). Before importing, you should verify that no one else is using the SMC files. If

Defining Your Synergy Methods

The Method Definition Utility

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 2-38](#).



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. See [example I on page 2-52](#).

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 2-43](#).

-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 [example J on page 2-52](#). For more information about this function, see [“Verifying Repository Structure Sizes and Enumerations” on page 2-41](#).

-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 examples below show various ways to start the MDU and run MDU utilities.

- A. The 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 2-48](#).

```
dbr DBLDIR:mdu
```

- B. 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
```

- C. 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
```

- D. The two examples below show how to start the MDU using SMC files on a remote machine that is running *xfServer*. 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

- E. 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
```

- F. 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 **rpmain.ism** and **rpstext.ism** in the DEV:[work] directory.

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

- G. In this example the contents of the SMC file located in c:\work\smc are exported 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
```

- H. In this example, the definitions in **CatalogA.xml** are imported 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
```

- I. 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
```

- J. 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 or 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 2-55](#) for OpenVMS). Running a comparison from Workbench enables you to verify the interfaces selected in a particular project (see [page 2-56](#)).



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 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.

`/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.

Defining Your Synergy Methods

The SMC/ELB Comparison Utility

`/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 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 4-2](#)
 - ▶ Java, see [page 7-3](#)
 - ▶ .NET, see [page 10-3](#)
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 3-2](#), “[Running *xfServerPlus* on UNIX](#)” on [page 3-8](#), or “[Running *xfServerPlus* on OpenVMS](#)” on [page 3-11](#). (Note: On OpenVMS, you must specify this account during installation. On Windows and UNIX, you can create the account after installation.)
 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 3-31](#).
 - ▶ 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](#).

Configuring and Running xfServerPlus

Running xfServerPlus

- ▶ 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. See [“Configuring Compression” on page 3-21](#).
- 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 3-18](#).
- 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 2-44](#).
- 6. Start *xfServerPlus*. See [“Running xfServerPlus on Windows” on page 3-2](#), [“Running xfServerPlus on UNIX” on page 3-8](#), or [“Running xfServerPlus on OpenVMS” on page 3-11](#).

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 3-5](#)).

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 Windows Control Panel, select Synergy Control Panel > Synergy Configuration Program) and go to the *xfServer/xfServerPlus* tab.
2. Click the Add *xfServerPlus* Service button.
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.

Description. (optional) Enter a descriptive string that will be added to the end of the command line when *xfServerPlus* is started. This string is then displayed at the end of the command line for the **db**s process in the Command Line column of the Processes tab in Windows Task Manager. (You may need to add the Command Line column to the Processes display.) If you have several *xfServerPlus* processes running at once, this enables you to distinguish among them. If the string contains %, it is replaced with the IP address of the *xfNetLink* client. (Do not use other %*letter* variables, and do not include quotation marks within the string.)

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

Configuring and Running xfServerPlus

Running xfServerPlus

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 3-43 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 4-2 on page 4-5](#).) 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 3-22 for details on this feature.



The encryption settings for the <Default> entry are used by both *xfServer* and *xfServerPlus*. For more information, 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. (Note that the resolved path will display rather than “DBLDIR”.)

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 and Functions” chapter of the *Synergy DBL 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 3-18](#) or [“Setting XFPL_SMCPATH on Windows” on page 2-45](#).



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 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.



For the complete list of **rsynd** options, see [“The rsynd Program”](#) in the *“Configuring xfServer”* chapter of the *Installation Configuration Guide*.

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

Configuring and Running xfServerPlus

Running xfServerPlus

You can override these defaults. See the syntax under [“Creating a second session” on page 3-6](#) 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, enter the following at the command line to start it:

```
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.

You can also start the service from the Windows Services console (the default display name is “Synergy/DE xfServerPlus”) or from the Synergy Configuration Program.

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. You can add a description to help distinguish the sessions.

- From the command line, register another instance of xfServerPlus, specifying a different service name, display name, and port number. You can use the **-text** option to add a description to help distinguish the sessions. For example:

```
rsynd -w -u xfspAcct/password -p 4567 -r -c xfspl_4567  
-d "xfServerPlus_4567" -text "Session two: %s"
```



For the complete list of **rsynd** options, 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. Use the **-c** option to specify a different service name; use the **-p** option to specify a non-default port.

- 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.

Configuring and Running xfServerPlus

Running xfServerPlus

- 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). You can specify a non-default port 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). You can specify a non-default port 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). You can specify a non-default port 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 (the default is 2356) for the additional session in the start-up syntax. You can use the **-text** option to add a description to help distinguish the sessions. For example:

```
rsynd -w -u xfspAcct -p 3356 -text "Session two: %s"
```



For the complete list of **rsynd** options, 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_XFNOCNN`.
- ▶ 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`.

- *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* .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.



Note that OpenVMS accounts have sections for “Default Privileges” and “Authorize Privileges”. The former lists those privileges that are always on, while the latter lists those that can be turned on but are off by default. The SHARE privilege must be defined in the “Default Privileges” section. (Usually, the same privileges are set in both sections.)

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 **SYS\$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.

Configuring and Running xfServerPlus

Running xfServerPlus

`/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 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.)

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 3-20.



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 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. 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 programs. (These are the methods in the *xfTest* interface in the distributed SMC.) These methods and the ELB must be present to use the test program.



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 2-38](#).

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 4-8](#)
- ▶ Java, see [page 9-9](#)
- ▶ .NET, see [page 12-9](#)

▶ 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 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 Developer 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 xfServerPlus sessions. Try logging on using that account. See “Running xfServerPlus on Windows” on page 3-2 .
	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 xfServerPlus 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 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 Windows” on page 3-2 .
	176 - 200	Rsynd licensing error.	Check the Windows application event log on the xfServerPlus machine for details on the exact error.

Configuring and Running xfServerPlus

xfServerPlus Status Codes

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 3-8.
	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 3-8.
	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 3-11.

Using the xfpl.ini File

The **xfpl.ini** file is read by **xfpl.dbr** each time an xServerPlus 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 xServerPlus runs in.

The default location for **xfpl.ini** is the DBLDIR directory. If you place the **xfpl.ini** file in a different location, set the XFPL_INIPATH environment variable to point to that location (see [page 3-18](#)).

If xServerPlus 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 xServerPlus logging is turned on (see [page 3-31](#)), errors will also be recorded in the xServerPlus log (**xfpl.log** by default).



See “[Appendix A: Configuration Settings](#)” 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 xServerPlus can reference only a single **xfpl.ini** file; however, if you have multiple instances of xServerPlus 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 Windows Control Panel, select Synergy Control Panel > 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 Windows Control Panel, select Synergy Control Panel > 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.

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 (or any environment variable) 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 for xfServer processes”](#) 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. To use compression, both xfServerPlus and xfNetLink must be version 8.1.5 or higher.



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 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 may even 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
```

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 3-33](#)) 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* client and *xfServerPlus*. *xfServerPlus* interfaces with a third-party library, OpenSSL, to provide SSL support for secure data transport.

For Synergy and .NET clients, both *xfServerPlus* and *xfNetLink* must be version 9.3 or higher; for Java, the minimum client version is 9.5.1a. 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 3-29](#) 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 3-23](#) for details.



Keep in mind that using encryption affects performance because data must be encrypted and decrypted on both sides of the *xfServerPlus*—*xfNetLink* connection. Consequently, slave encryption affords better performance than master encryption. The cipher negotiation itself also takes time. You will get better performance if your client makes a connection, performs all the necessary method calls, and then disconnects. If you connect, make a call, and disconnect, repeating this process for each call, performance may be degraded because the cipher negotiation has to be performed on each call. If you are using a Java or .NET client, you may wish to use pooling to further improve performance, as the connections (and, consequently, the cipher negotiations) are made when the pool is first populated, rather than with each call. By using pooling in combination with slave encryption, you can achieve both security and a high level of performance.

When encryption is enabled, you'll see an entry in the *xfServerPlus* log. If XFPL_SESS_INFO is set to ALL (see [page 3-33](#)), 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_RANDOM 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_RANDOM](#) in the “Environment Variables” chapter of *Environment Variables & 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.
- ▶ 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 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, the OpenSSL libraries must be located on the *xfServerPlus* machine in the `dbl\bin` directory (as explained in [KB 100001979](#)). 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`
 - ▶ Oracle Solaris 64-bit SPARC (320): `/usr/lib/64`
 - ▶ Oracle 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, `SYNSSLLIB` is set in `SYSS$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 “`SYNSSLLIB` not set”.

3. Create a certificate (.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 [“Requesting a certificate”](#) in the “Synergy HTTP Document Transport API” chapter of the *Synergy DBL 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 Synergy Machine for Encryption

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

- ▶ On Windows machines, install OpenSSL in the db\bin directory.
- ▶ 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 3-24](#).

Setting up the xfNetLink Java Machine for Encryption

Java encryption does not use the **.pem** file directly. Rather, it requires that the certificates be placed in a keystore file. A default keystore file, **cacerts**, is distributed with the JRE. You can use the **genCert** utility (see below) to add additional certificates to this file.

The **genCert** utility creates a copy of the distributed **cacerts** file and adds the certificates from the **.pem** file on the *xfServerPlus* machine to it. We recommend you use this new file and call it something other than “cacerts” because the **cacerts** file is replaced whenever Java is upgraded.

Configuring and Running xfServerPlus

Using Encryption

Once *xfServerPlus* has been started with encryption enabled, do the following on the *xfNetLink* Java machine:

1. Run the **genCert** utility to create the certificate file. See below for complete syntax. For example,

```
java genCert -h myServer -p 3535
```

2. Set the `xf_SSLCertFile` and `xf_SSLPassword` settings in the *xfNetLink* Java properties file to specify the filename and password for the certificate file created in step 1. See [“Specifying Encryption Options” on page 8-9](#). Alternatively, you may use the `setSSLCertFile()` and `setSSLPassword()` methods to specify this information at runtime. See the method reference on [page 8-35](#).

The genCert Utility

The **genCert** utility, which is distributed in the `xfNLJava` directory, assists you in setting up encryption on your *xfNetLink* Java machine. *xfServerPlus* must be running with encryption enabled before you can run **genCert**.

Syntax `java genCert [-c filename] [-s password] [-n filename] -h host [-p port] [-?]`

Arguments -c filename

(optional) The name and path of the base Java certificate file. The default is *java.home*\lib\security\cacerts, where *java.home* is the JRE installation directory.

-s password

(optional) The password for the base certificate file. If not passed, the default is “changeit”. (Note that this is the password for the **cacerts** file distributed with the JRE; your system administrator may have changed it.)

-n filename

(optional) The name and path of the new certificate file to put the generated certificates in. If not passed, the default is **newcerts**, and the file is placed in the current working directory.

-h host

The name of the *xfServerPlus* machine to connect to. *xfServerPlus* must be running with encryption enabled.

-p port

(optional) The port that *xfServerPlus* is running on. If not passed, the default is 2356.

-?

Display usage message.

Usage & Examples When you run the **genCert** utility, you will see output similar to that shown below. The utility outputs the parameters it is using, followed by the certificates found in the **.pem** file on the *xfServerPlus* machine. Notice that the certificates are numbered. At the end, you'll be prompted to enter the number of the certificate that you want added to the keystore file (the default is 1), and the file will be created. The new certificate file has the same password as the original file. Type **q** at the prompt to quit without creating the new certificate file.

The example below uses the default **cacerts** file and password, and then specifies a name for the new certificate file, along with the *xfServerPlus* machine and port to connect to.

```
java genCert -n myCertFile -h myServer -p 3535
```

```
Checking for valid certificate in base certificate file.
```

```
Base certificate file = c:\java\jdk6_12\jre\lib\security\cacerts
```

```
Base certificate password = changeit
```

```
New certificate file = myCertFile
```

```
xfServerPlus Host = myServer
```

```
xfServerPlus Port = 3535
```

```
Loading KeyStore c:\java\jdk6_12\jre\lib\security\cacerts...
```

```
Server sent 2 certificate(s):
```

```
1 Subject EMAILADDRESS= user@synergex.com, CN=myServer, OU=SDE,
O=Synergex, L=Sacramento, ST=California, C=CA
Issuer EMAILADDRESS= user@synergex.com, CN=myServer, OU=SDE,
O=Synergex, L=Sacramento, ST=California, C=US
sha1 c4 a3 f8 63 85 84 b6 cb b1 63 65 9e ca e5 bf 34 8f 5f e8
md5 26 87 6c 34 77 ac 8d e9 39 27 99 f1 0a fc 97 5f
```

```
2 Subject EMAILADDRESS= user@synergex.com, CN=myServer, OU=SDE,
O=Synergex, L=Sacramento, ST=California, C=US
Issuer EMAILADDRESS= user@synergex.com, CN=myServer, OU=SDE,
O=Synergex, L=Sacramento, ST=California, C=US
sha1 07 d2 b4 e7 d7 7a 9f 09 44 88 3c 50 1c 20 5f 90 0d cf dd
md5 55 1d f6 13 3b 6c cd e3 7c a6 60 75 93 7f e4 a8
```

```
Enter certificate to add to trusted keystore or 'q' to quit: [1]
```

```
1
```

```
Certificate added
```

Setting up the xfNetLink .NET Machine for Encryption

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. The dialog for specifying the SSL cipher suite order is slightly different for Vista/2008 than for Windows 7/8.

On Vista and Server 2008,

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 enables 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 12-7](#).
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.

On Windows 7 and 8,

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 dialog.
4. Select Enabled and click the Apply button. (The default is Not Configured, which enables some cipher suites, but does not necessarily ensure that you have the correct suites enabled.)

You will see the enabled suites listed below in the SSL Cipher Suites field in the Options pane. Although only a limited number of characters display, the field contains a great number of cipher suites by default. You can copy the contents and paste them into a text editor to see the default cipher suite order.

5. Look at the Help in the pane on the right 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 12-7](#).
6. When you are done, click OK in the SSL Cipher Suite Order 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_SETTRMTFNC 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 6-23](#) and [RX_SETTRMTFNC on page 6-16](#) 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).

Configuring and Running xfServerPlus

Using Server-Side Logging

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 Java

To indicate which methods should be encrypted when you are using xfNetLink Java, 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 JAR file.

When you generate classes, a new method, `setEncryptedMethod(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.

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 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 3-17](#)). 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:\TempFiles\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 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 that 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

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.

Configuring and Running xfServerPlus

Using Server-Side Logging

XFPL_SESS_INFO (continued)		
Value	Information logged	Description
CRITICAL or ALL	xfServerPlusRemoteSession connected to	The type of client xfServerPlus is connected to: Synergy, Java, or .NET.
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 3-40 for the correspondence between the protocol version and software version.
ALL	Compression	Indicates whether xfServerPlus compression is on or off.
ALL	Average compression	If compression is on, displays the average percentage of compression for data sent and received during this session.
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. “000007F4” is the session ID.

```
000007F4: xfServerPlusRemoteSession 10.1.1 started: 31-JUL-2012
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 10.1.1 stopped: 31-JUL-2012
16:35:53
```

The example below shows the type of information that is recorded when XFPL_SESS_INFO is set to ALL.

```
00000308: xfServerPlusRemoteSession 10.1.1 started: 31-JUL-2012
16:29:56
00000308: xfNetLink ip address = 111.22.33.44, 6F16212C
00000308: Server Protocol Version: 6
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 10.1.1 stopped: 31-JUL-2012
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

Configuring and Running xfServerPlus

Using Server-Side Logging

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 3-34 .
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.
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 xfServerPlus.
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 AT = autotime 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.

XFPL_FUNC_INFO (continued)		
Value	Information logged	Description
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.

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 31-JUL-2012 16:25:57
00000418: Method id: xfpl_tst1
00000418: Function: function_one
00000418: ELB: DBLDIR:xfpl_tst
00000418: Packet returned at 31-JUL-2012 16:25:57:990000
00000418: Packet received at 31-JUL-2012 16:25:58:991000

```

The example below shows the type of information that is recorded when XFPL_FUNC_INFO is set to ALL.

```

000003E0: Packet received at 31-JUL-2012 16:16:50:146000
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 31-JUL-2012 16:16:50:167000
000003E0: **** Outgoing Parameters ****

```

Configuring and Running xfServerPlus

Using Server-Side Logging

```
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 31-JUL-2012 16:16:50:209000
000003E0: Packet received at 31-JUL-2012 16:16:51:229000
```

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 3-33 .
Information that is recorded when XFPL_FUNC_INFO is set to ALL	Function information; see page 3-36 .
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: 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.

XFPL_DEBUG (continued)	
Information logged	Description
Not returning parameter [n]	<p>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:</p> <p>AL = alpha AT = autotime BI = binary (handle) DEC = decimal ENUM = enumeration HA = memory handle IMPDEC = implied-decimal INT = integer SS = System.String STR = structure</p> <p>The log also includes parameter and returning parameter information, recorded as part of the function information.</p>

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 10.1.1 started: 31-JUL-2012
12:40:13
00000D00: Command line = {C:\Program Files\Synergex\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: 6
00000D00: Packet = { 31-JUL-2012 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 31-JUL-2012 12:40:14:411000
00000D00: Method id: xfpl_tst6
00000D00: Function: function_ten
00000D00: ELB: DBLDIR:xfpl_tst
00000D00: **** Incoming Parameters ****
00000D00: Parameter[1] = Type: (AL), Size: (5)
00000D00: Data [abcde]

```

Configuring and Running xfServerPlus

Using Server-Side Logging

```
00000D00: Parameter[2] = Type: (AL), Size: (5)
00000D00: Data [54321]
00000D00: **** End of Incoming Parameters ****
00000D00: Function function_ten called at 31-JUL-2012
12:40:14:502000
00000D00: Function function_ten returned at 31-JUL-2012
12:40:14:531000
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 31-JUL-2012 12:40:14:573000
00000D00: Packet =
{Rxfp1_tst6;002;000DE9#123456789;002AL4#back;}
00000D00: Packet = {S;0;;}
00000D00: Packet received at 31-JUL-2012 12:40:14:640000
00000D00: xfServerPlusRemoteSession 10.1.1 stopped: 31-JUL-2012
12:40:14
```

Protocol version

The protocol version for *xfServerPlus* is recorded in the *xfServerPlus* log and may occasionally appear in an error message. The table below shows the correspondence between protocol version and software version.

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, 9.5, 10

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 routine execution continues. The error message text shown in the table may be accompanied by additional detail information.

You may also see regular Synergy DBL errors with additional detail information. See the “[Error Messages](#)” chapter in *Synergy 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
2016	Global not found: <i>global_name</i>
2017	File not found: <i>filename</i>

Configuring and Running xfServerPlus

Using Server-Side Logging

xfServerPlus Error Messages (continued)	
Error number	Error message
2018	Internal routine not found
2019	Secure communications required
2020	Old ELB file format detected—relink
2021	Bad ELB detected
2022	ELB file built with opposite 'endian'
2023	ELB file built with opposite bit size
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 **db**s 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 (**db**s). This means that your environment may change, because **dbr** always reads the **synergy.ini** file, whereas **db**s 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 **db**s, see “The service runtimes” in the “Building and Running Traditional Synergy Applications” chapter of *Synergy 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 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.

Configuring and Running xfServerPlus

Debugging Your Remote Synergy Routines

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 5-4](#)
- ▶ Java, see [page 9-10](#)
- ▶ .NET, see [page 12-10](#)

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 Traditional Synergy Programs” chapter of *Synergy 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 3-43](#).

If your Java 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 3-8](#). For complete rsynd syntax, see [“The rsynd Program”](#) in the “Configuring xfServer” chapter of the *Installation Configuration Guide*.

Configuring and Running xfServerPlus

Debugging Your Remote Synergy Routines

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 Traditional Synergy Programs](#)" chapter in *Synergy 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 3-18](#).
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 Tools*. For information on importing methods, see [“Importing and Exporting Methods” on page 2-38](#).
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 2-44](#).
6. Start xfServerPlus. See [“Running xfServerPlus” on page 3-2](#) 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 4-2](#)
- ▶ Java, see [page 8-1](#)
- ▶ .NET, see [page 11-1](#)

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.

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](#).

Configuring and Running xfServerPlus

Configuring xfServerPlus for Remote Data Access

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 **setruser -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 3-50](#). In [step 6](#), rather than pasting the string, just enter the user name for the variable value.

► 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 3-2](#) 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
```

Configuring and Running xfServerPlus

Configuring xfServerPlus for Remote Data Access

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

Configuring & Testing *xfNetLink* Synergy

xfNetLink Synergy Edition provides distributed access to Synergy logic and Synergy data (via access to Synergy logic) from within Synergy applications. *xfNetLink* Synergy is a set of routines included in the Synergy runtime that can be used to call routines that are executed on a remote machine running *xfServerPlus*.

This chapter gives an overview of the tasks you must perform to set up and use *xfNetLink* Synergy Edition, and explains how to configure *xfNetLink* Synergy and run the *xfNetLink* Synergy test program.

System Overview

Figure 4-1 shows the primary components of a distributed Synergy application using *xfServerPlus* and *xfNetLink* Synergy. The diagram describes two logical machines:

- ▶ A Synergy client running a Synergy application that includes the *xfNetLink* Synergy API and contains the user interface. A single client session may create multiple sessions on one server machine.
- ▶ A Synergy server running *xfServerPlus*, 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). 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 *xfServerPlus* license

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.

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

Configuring & Testing x/NetLink Synergy

The Big Picture

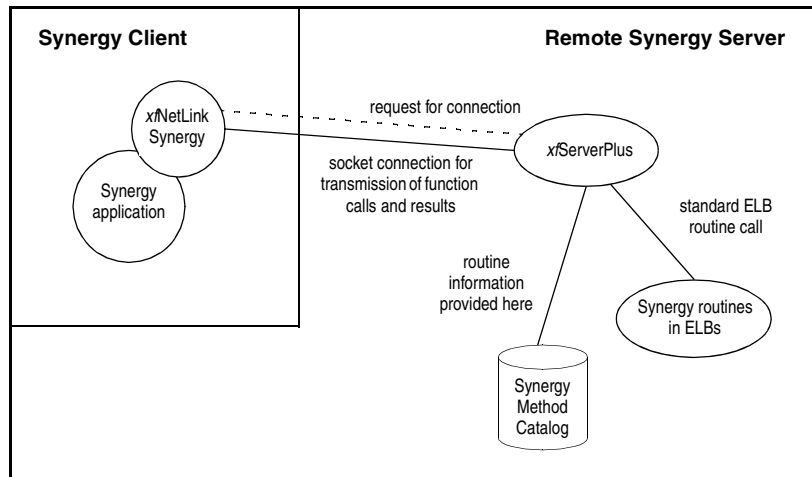


Figure 4-1. x/ServerPlus with x/NetLink Synergy.

client and a dedicated x/ServerPlus 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. Once the connection has been established, your remote routines are available to the client application.

The Big Picture

To successfully set up and run your x/NetLink Synergy–x/ServerPlus system, you need to do complete the following steps.

1. Install the necessary Synergy software.
 - Install x/ServerPlus on the Synergy server machine. For detailed steps on setting up x/ServerPlus, see [“The Big Picture” on page 3-1](#).
 - Install x/NetLink Synergy on the client machine. For development purposes, you’ll need to install Professional Series Development Environment. However, because the x/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 Synergy DBL (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 *xfServerPlus* machine to run *xfServerPlus* sessions. Give this account privileges that are consistent with your policy for accessing Synergy routines remotely. See [“Running xfServerPlus” on page 3-2](#).
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.
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. You may also need to set other options in the **xfpl.ini** file; see [“Appendix A: Configuration Settings”](#) 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 4-4](#).
7. Create the user interface for your Synergy client. Include code that requests a remote execution session using the *xfNetLink Synergy* API. See [chapter 5](#) for “how-to” information about using the API and [chapter 6](#) for the API routines.
8. Start *xfServerPlus* (**rsynd**) with remote execution enabled. See [“Running xfServerPlus” on page 3-2](#).
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: Configuration Settings](#)” for complete listings of all configuration settings for xfServerPlus and xfNetLink.

Variables for Configuring xfNetLink Synergy	
Variable	Description
XF_REMOTE_HOST	Machine where xfServerPlus is running
XF_REMOTE_PORT	Port that xfServerPlus 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 4-2](#)) 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.

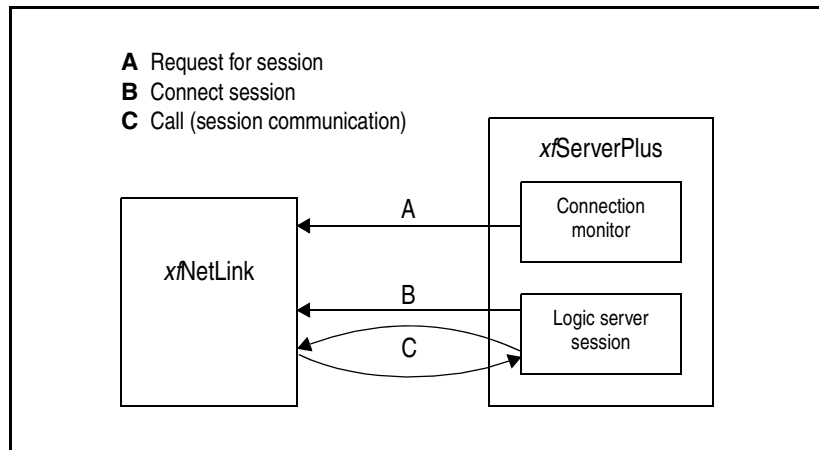


Figure 4-2. *xfNetLink* time-outs.

Connect session time-out

Once the initial socket communication is established ('A' in [figure 4-2](#)), the connection monitor signals the logic server to start a session. The connect session time-out ('B' in [figure 4-2](#)) 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 5-5](#)).

► 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 4-2](#)) 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 xfNetLink and xfServerPlus. 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 6-15](#)); this affects only the current session.

You can also set a call time-out value for *xfServerPlus*. 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 3-30](#).

► 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 2-38](#).

► 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 3-14](#) 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 *xfNetLink* 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 5-3](#).

Making Remote Calls with `%RXSUBR`

For a complete code sample, see [“Appendix C: *xfNetLink* Synergy Sample Code”](#). For details on the `RX_xxx` routines, see [chapter 6](#).

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 *xfNetLink* Synergy” on page 4-4](#).) For example:

```
status = %rx_start_remote(netid, ip, port)
```

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

Calling Synergy Routines Remotely from Synergy

Making Remote Calls

(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 5-3.](#))

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 arrayed field in Synergy DBL 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 1-13](#) and [“Passing Arrays Larger Than 64K” on page 1-18](#). For instructions on using the RCB routines and the RCB routine syntax, see the [“Synergy Routine Call Block API”](#) chapter of the *Synergy DBL 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 3-30](#).) 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 Troubleshooting Techniques

There are two categories of *xfServerPlus* errors.

- ▶ Non-fatal errors that *xfServerPlus* can trap and for which it can produce a meaningful diagnostic
- ▶ Fatal errors that *xfServerPlus* cannot trap and which cause abnormal termination of the remote execution session

In both cases, *xfServerPlus* 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 6-26](#)).

In the event of a non-fatal error, the *xfServerPlus* session remains available for future calls. Detailed error information can be retrieved by calling `RX_GET_ERRINFO` (see [page 6-7](#)).

In the event of a fatal error, the session is aborted. Traceback information can be retrieved by calling `RX_GET_HALTINFO` (see [page 6-9](#)).

You may also see *xfServerPlus* status codes returned to the client. See the “*xfServerPlus* Status Codes” table on [page 3-15](#).

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 4-8](#) and “*Testing xfServerPlus*” on [page 3-14](#).

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 5-8](#).



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 3-45](#) 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.)

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:
2. When the IP and port display on the screen, write them down. You'll need them in [step 4](#). 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))
```



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 4-6](#)) 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*.

Calling Synergy Routines Remotely from Synergy Troubleshooting Techniques

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 *x/ServerPlus*, start **xfpl.dbr**, and pass the IP and port to *x/ServerPlus*. Type the alpha characters in the IP address in uppercase.

```
dbr -d xfpl hexadecimal_IP port
```

For example:

```
dbr -d xfpl 6F16212C 1082
```

x/ServerPlus 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
```

x/ServerPlus 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 *x/ServerPlus* 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 *x/ServerPlus*. 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 Traditional Synergy Programs](#)” chapter in *Synergy 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
```

You'll see output similar to the following:

```
*****
***  DEBUG 10.1.1    ***
BREAK AT 152 IN XFPL (LAUNCHER.DBL;6) ON ENTRY
%DBG-E-Could not open source file "LAUNCHER.DBL;1"
Dbldbgt>
*****
```



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 Traditional Synergy Programs” chapter of *Synergy 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 Traditional Synergy Programs” chapter in *Synergy 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 5-4).

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 3-30.

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 XFNL_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 *xfServerPlus* log shows you the packets that are received by and sent from *xfServerPlus*. To record packets in the *xfServerPlus* log, set the XFPL_DEBUG option to ON in the **xfpl.ini** file. See [XFPL_DEBUG on page 3-38](#) 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` 6-2
`%RX_DEBUG_INIT` 6-4
`%RX_DEBUG_START` 6-5
`RX_GET_ERRINFO` 6-7
`RX_GET_HALTINFO` 6-9
`%RX_RMT_ENDIAN` 6-11
`%RX_RMT_INTSIZE` 6-12
`%RX_RMT_OS` 6-13
`%RX_RMT_SYSINFO` 6-14
`%RX_RMT_TIMEOUT` 6-15
`RX_SETRMTFNC` 6-16
`RX_SHUTDOWN_REMOTE` 6-17
`%RX_START_REMOTE` 6-18
`%RXSUBR` 6-23

%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 5-5](#) 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 “%RX_START_REMOTE Status Codes” table on page 6-20.

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 “Synergy Runtime Errors Signaled by %RXSUBR” table on page 6-26 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 XFNLN_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 4-2 on page 4-5](#).)

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 6-15](#)).



For more information on setting time-outs for *xfNetLink*, see [“Specifying Time-out Values” on page 4-5](#).

`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 4-2 on page 4-5](#).) Note that the time-out for the request for session from the connection monitor is 2 minutes (‘A’ in [figure 4-2](#)): 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 5-5](#) 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
```

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
.
```

xfNetLink Synergy API

RX_GET_ERRINFO

```
.
.
handle_parmerr,
    xcall RX_GET_ERRINFO(netid, errinfoec)
    xcall handle_parmerr(errinfoec)
    xreturn

handle_other,
    xcall RX_GET_ERRINFO(netid, errinfoec)
    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 “[Synergy Runtime Errors Signaled by %RXSUBR](#)” table on page 6-26 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 “[Synergy Runtime Errors Signaled by %RXSUBR](#)” table on page 6-26 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 “[Synergy Runtime Errors Signaled by %RXSUBR](#)” table on page 6-26 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 “[Synergy Runtime Errors Signaled by %RXSUBR](#)” table on page 6-26 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 “%RX_START_REMOTE Status Codes” table on page 6-20 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 “Synergy Runtime Errors Signaled by %RXSUBR” table on page 6-26 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 4-2 on page 4-5.)

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 4-5).

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 3-22](#) for more information on encryption.

See the [“Synergy Routine Call Block API”](#) chapter of the *Synergy DBL 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 “%RX_START_REMOTE Status Codes” table on page 6-20 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 4-2 on page 4-5](#).)

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 6-15](#)).



For more information on setting time-outs for *xfNetLink*, see “Specifying Time-out Values” on [page 4-5](#).

`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 4-2 on page 4-5](#).) Note that the time-out for the request for session from the connection monitor is 2 minutes (‘A’ in [figure 4-2](#)): 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 Sample Code](#)”.

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 2-44 . 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_NOXF SPL	<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 3-22](#) 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 6-26](#)) 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 6-26](#)). To get more detailed information about a particular error, call `RX_GET_ERRINFO` (see [page 6-7](#)) or `RX_GET_HALTINFO` (see [page 6-9](#)), 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: xfNetLink Synergy Sample Code](#)” 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 4-5) or optimize the called routine. If the problem persists, check with your system administrator. You can use %RX_CONTINUE (see page 6-2) 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.
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.

Synergy Runtime Errors Signaled by %RXSUBR (continued)			
Number	Runtime error	Caused by	What to do
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.
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.

Synergy Runtime Errors Signaled by %RXSUBR (continued)			
Number	Runtime error	Caused by	What to do
592	\$ERR_XFINCALL	Remote call already in progress	Use %RX_CONTINUE (see page 6-2) 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 6-23 for information on /encrypt and "Using Encryption" on page 3-22 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 6-23 for information on /encrypt and "Using Encryption" on page 3-22 for information on encryption in general.

Part III: *xf*NetLink Java Edition ▶

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 generated class wrappers use the *xfNetLink* Java classes internally to connect to *xfServerPlus*. The JAR file that you generate can be used in any Java client application to remotely access your Synergy business logic on the *xfServerPlus* machine.

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 1.5 or higher, available free from <http://www.oracle.com/technetwork/java/index.html>. See the release notes (REL_XFNJ.TXT) for more information.
- (optional) Java development environment, such as [Eclipse](#) or [NetBeans](#)
- Web server and JSP/Servlet container if you plan to develop a web application



Starting with version 9.5.1a, *xfNetLink* Java is not supported on SCO OpenServer (system code 003) or HP Tru64 UNIX (system code 021) because these platforms do not support Java 1.5.

System Overview

[Figure 7-1](#) shows the primary components of a distributed application that accesses Synergy code from a Java client. This diagram describes two machines:

- A client machine running *xfNetLink* Java, the Java Runtime Environment (JRE), and an application that uses the JAR file built from Synergy methods. If you're developing a two-tier system with a Java client application, the client is the end-user's machine. If you're developing a three-tier system with a web client application using JavaServer Pages, the client is the web server machine, which is also the location of your HTML and JSP pages and the JSP/servlet container.

Creating Java Class Wrappers

System Overview

- A Synergy server running *xfServerPlus*, 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 *xfServerPlus* license.

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.

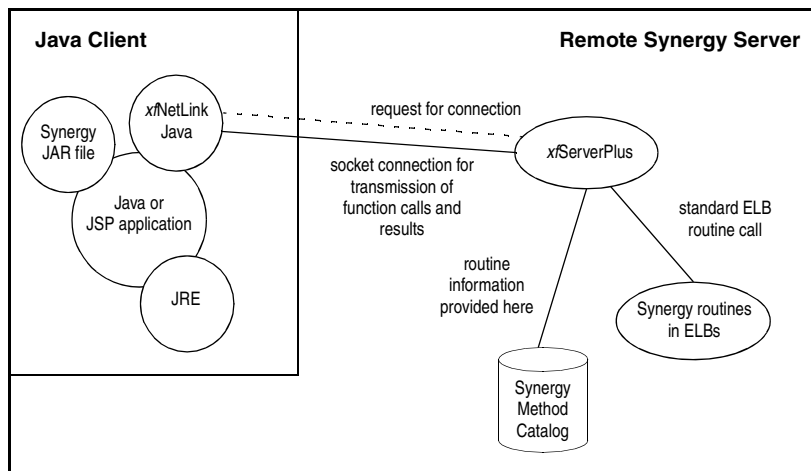


Figure 7-1. Accessing Synergy from Java.

The Big Picture

This section lists all the steps that need to be completed to successfully create a distributed system using *xfNetLink* Java. 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 1.5 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 distribution. After installing, set your classpath. See [“Setting the Classpath” on page 7-5](#).
 - ▶ 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 DBL 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_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. You may also need to set other options in the **xfpl.ini** file; see [“Appendix A: Configuration Settings”](#) for a complete list of **xfpl.ini** configuration settings.

Creating Java Class Wrappers

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.

6. Create a user account on the *xfServerPlus* machine to run *xfServerPlus* sessions, and then start *xfServerPlus*. See [“Running xfServerPlus” on page 3-2](#).
7. Generate the class wrappers:
 - ▶ If you’re using Workbench, create a Synergy/DE Java Component project. Specify the component information: a name and location for the JAR file, the Java version you want to target, the location of the SMC and repository files, the package name, the interfaces you want to include, and (if desired) alternate names for those interfaces. Then, use the menu option to generate class wrappers. See [“Creating a Synergy/DE Java Component Project” on page 7-6](#) and [“Generating Java Class Wrappers” on page 7-9](#).
 - ▶ If you’re using the command line, run the **genxml** utility to create an XML file, and then run **genjava** to create the Java class wrappers. See [“Creating a Java JAR File from the Command Line” on page 7-11](#).
8. (optional) Modify the generated Java source files if necessary. See [“Editing the Java Source Files” on page 7-20](#).
9. Build the JAR file:
 - ▶ 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 7-10](#).
 - ▶ If you’re using the command line, 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 7-16](#).
10. 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 8-1](#) and [“Using Your JAR File” on page 8-10](#).

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. The classpath also needs to include the **xercesImpl.jar** and **xml-apis.jar** files (for the XML parser required by the **genjava** utility) and the *xfNetLink* Java installation directory. The classpath setting must be a system-wide setting, not just a user login setting.

For example, on Windows:

```
classpath=c:\Program Files\Synergex\xfNLJava;c:\Program Files\Synergex\xfNLJava\xfnljav.jar;c:\Program Files\Synergex\xfNLJava\xercesImpl.jar;c:\Program Files\Synergex\xfNLJava\xml-apis.jar;.
```



When you deploy your application at a customer site, you do not need to include the **xercesImpl.jar** file or **xml-apis.jar** file 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 7-11](#).

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 see the Synergy/DE Java Component node.)
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*.

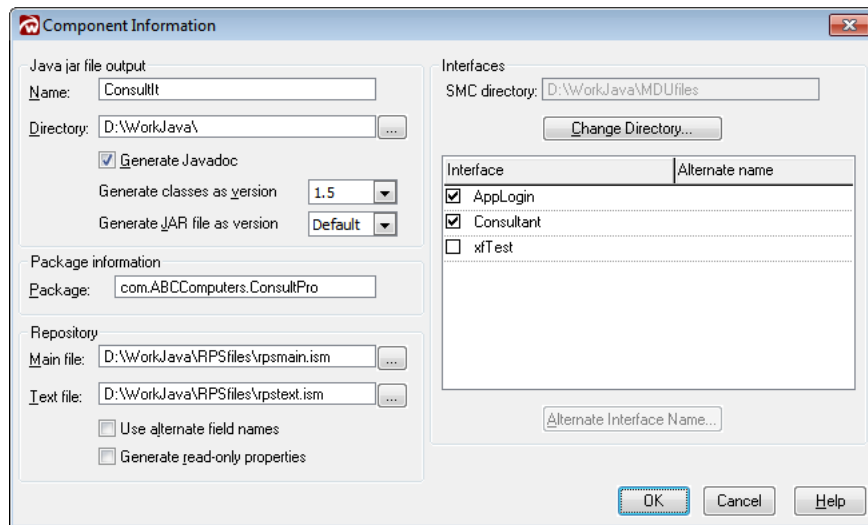


Figure 7-2. The Component Information dialog box for a Synergy/DE Java Component project.

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., MYDIR:). 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.

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 7-20](#) for additional information.

Generate classes as version. Indicate the Java compatibility desired for the generated classes. If you select 1.2, the classes will be generated as they were in x/NetLink Java version 9.5.1 and earlier, and they will be compatible with JRE 1.2 through 1.4. If you select 1.5, the generated classes will be compatible with JRE 1.5, and various new x/NetLink Java features (introduced in version 9.5.1a and later) will be available. These include encryption, enumerations, type coercion, and support for Boolean and binary structure fields. In addition, when you select 1.5, default type mappings change (see [“Appendix B: Data Type Mapping”](#)) and ArrayLists use generics.

Generate JAR file as version. Indicate the version of the JRE that the JAR file should be built to target. Select Default to use the version of Java installed on the machine where the JAR file is built.

Package. 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. 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.

Creating Java Class Wrappers

Creating a Java JAR File in Workbench

Repository text file. 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.

Generate read-only properties. Indicate whether you want fields flagged as read-only in Repository to be generated as read-only properties in the structure classes. These properties will have a “get” method but no “set” method. If you are planning to use the classes in your JAR file as JavaBeans, note that JavaBeans require that properties be accessible with both get and set methods.

SMC directory. This field displays the path for the Synergy Method Catalog that this JAR file uses. The default is XFPL_SMCPATH; if it is not set, 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. 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](#) for a full explanation.

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 Java class wrappers, select Build > Generate Class Wrappers. 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.

This command will do the following:

- ▶ Run the SMC/ELB Comparison utility (**smc_elb.exe**; see [page 2-53](#) for more information).
 - ▶ Create a Java source file for each interface you selected and for each structure, group, and enumeration within the selected interfaces.
 - ▶ Create a manifest file that names the classes in the JAR file and indicates they can be JavaBeans™.
 - ▶ Create a batch file to be used later to build 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.
2. Edit the files, if necessary, before building the JAR file. See [“Editing the Java Source Files” on page 7-20](#).

Building the JAR File



If you are using Java 1.7 and targeting an earlier version of Java (1.5 or 1.6), you must set the XFBOOTCLASSPATH and XFEXTDIRS environment variables *before* building the JAR file. If either is missing or not set to a valid location, the compile will fail.

Set XFBOOTCLASSPATH to the complete path of the **rt.jar** file, usually *java.home\lib\rt.jar*. (For example, c:\Program Files\Java\jdk1.6.0_23\jre\lib\rt.jar.)

Set XFEXTDIRS to the directory that contains the other classes and jar files that are required by your application, usually *java.home\lib*. (For example, c:\Program Files\Java\jdk1.6.0_23\jre\ lib.)

There are a number of ways to set environment variables so that they may be used by Workbench. See [“Defining the startup environment”](#) in the “Developing Your Application in Workbench” chapter of *Getting Started with Synergy/DE* for details.

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” option was selected 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 an XML file. See [“The genxml Utility” on page 7-11](#).
2. Use the **genjava** utility to create the Java classes. See [“The genjava Utility” on page 7-14](#).
3. Edit the Java source files if necessary. See [“Editing the Java Source Files” on page 7-20](#).
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 7-16](#).

The genxml Utility

The **genxml** utility creates an 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 2-41](#).) Failure to do so can cause errors at runtime because the structure information in the component, which was pulled from the repository, will differ from that in the SMC.

Creating Java Class Wrappers

Creating a Java JAR File from the Command Line

Syntax `dbf 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 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 7-13](#).

`-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 an XML file named **ConsultIt.xml**, which 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
```

Creating Java Class Wrappers

Creating a Java JAR File from the Command Line

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, group, and enumeration 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] [-c version] [-t version] [-ro] [-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 7-20.](#))

`-c version`

(optional) Indicates the Java compatibility desired for the generated classes. Valid values for *version* are 1.2 and 1.5. If you specify 1.2, the classes will be generated as they were in *x/fNetLink* Java version 9.5.1 and earlier, and they will be compatible with JRE 1.2 through 1.4. If you specify 1.5, the generated classes will be compatible with JRE 1.5, and various new *x/fNetLink* Java features (introduced in version 9.5.1a and later) will be available. These include encryption, enumerations, type coercion, and support for Boolean and binary structure fields. In addition, when you specify 1.5, default type mappings change (see [“Appendix B: Data Type Mapping”](#)) and ArrayLists use generics. If `-c` is not specified, the default is 1.5.

`-t version`

(optional) Indicates the version of the JRE that the JAR file should be built to target. (This option adds the `-target` option to the command line in the batch file.) Valid values for *version* are 1.5 and 1.6. If `-t` is not specified, the default is the version of Java installed on the machine where the JAR file is built.

`-ro`

(optional) Indicates that you want fields flagged as read-only in Repository to be generated as read-only properties in the structure classes. These properties will have a “get” method but no “set” method. If you are planning to use the classes in your JAR file as JavaBeans, note that JavaBeans require that properties be accessible with both get and set methods.

Creating Java Class Wrappers

Creating a Java JAR File from the Command Line

-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 **.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**, with the generated **.java** files placed in the **ConsultPro** directory.

```
java genjava -f c:\work\ConsultIt.xml  
-p com.ABCComputers.ConsultPro -d c:\work -j -c 1.5 -t 1.5
```

Building the JAR File

If you need to edit the **.java** source files, do so before building the JAR file. See [“Editing the Java Source Files” on page 7-20](#).



If you are using Java 1.7 and targeting an earlier version of Java (1.5 or 1.6), you must set the **XFBOOTCLASSPATH** and **XFEXTDIRS** environment variables *before* running the batch file. If either is missing or not set to a valid location, the compile will fail.

Set **XFBOOTCLASSPATH** to the complete path of the **rt.jar** file, usually *java.home\lib\rt.jar*. (For example, *c:\Program Files\Java\jdk1.6.0_23\jre\lib\rt.jar*.)

Set **XFEXTDIRS** to the directory that contains the other classes and jar files that are required by your application, usually *java.home/lib*. (For example, *usr/java/jdk1.6.0_23/jre/lib*.)

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 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.

Run the batch file. For example, to continue with our previous scenario:

```
c:\work\ConsultIt
```

The JAR file 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 7-20](#) and [“Deploying the Client” on page 8-1](#) 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, group, and enumeration within the selected interfaces. These generated classes use the *xfNetLink* Java classes internally to connect with *xfServerPlus* and pass data.

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>getSSLCertFile()</code>
<code>disconnect()</code>	<code>setSSLCertFile()</code>
<code>debugInit()</code>	<code>getSSLPassword()</code>
<code>debugStart()</code>	<code>setSSLPassword()</code>
<code>getxfPort()</code>	<code>getUserString()</code>
<code>setxfPort()</code>	<code>setUserString()</code>
<code>getxfHost()</code>	<code>setxfCallTimeout()</code>
<code>setxfHost()</code>	<code>getConnect()</code>
<code>getxfLogfile()</code>	<code>shareConnect()</code>
<code>setxfLogfile()</code>	<code>getPoolName()</code>
<code>setxfLogging()</code>	<code>usePool()</code>
<code>getSynergyWebProxy()</code>	

These methods are used to connect to and disconnect from *xfServerPlus* 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 *xfServerPlus* log, and for specifying client-side logging options. See the method reference on [page 8-35](#) 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. 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.



If **genjava** was run with the **-ro** option (or the “Generate read-only properties” option was selected in Workbench), structure fields that are marked read-only in the repository will be generated with get methods but no set methods.

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 8-14](#).

Enumeration Classes

Enumeration classes are built from your enumeration definitions in the repository. There will be a `.java` 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 8-16](#).

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 7-20.



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 the 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 Oracle website for more information: <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.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 2-20](#).
- Edit the generated source files (.java files) to add Javadoc comments or to add additional formatting tags to create the desired output. This method is not recommended, because any changes you make to the .java files will be lost if you regenerate class wrappers. See [“To manually add Javadoc comments” on page 7-22](#) for instructions.

The descriptions in the SMC are included in the generated files as Javadoc comments when you generate class wrappers. 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 descriptions in the repository are used as the Javadoc comments for the set and get property methods in the generated classes. 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 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 8-1](#) 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 *xfNetLink* Java properties file, and then use the JAR file. It also includes information on Java connection pooling, a method reference for the utility methods included in your class wrappers, and a class reference for classes in the *Synergex.util* package.

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 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 3-47](#).

Deploying the Client

1. Verify that JDK 1.5 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 version of the Java Runtime Environment that is compatible with your application.)
2. Install *xfNetLink* 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 7-5](#).
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

Deploying Your Distributed Application

5. If you created Javadoc, copy the Javadoc HTML files to the client machine. You will need to distribute **index.html** and at least one of the **allclasses*.html** files, which will be located in the package directory, along with 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.

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 8-24](#) 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: Configuration Settings”](#) 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. You can also set logging and encryption options, and various time-out values.

Using a properties file is optional in most cases. 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 8-5](#).) 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 8-27](#) 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 <i>xfServerPlus</i> is running	8-6
xf_RemotePort	Port that <i>xfServerPlus</i> is running on	8-6
xf_DebugOutput	Boolean value that turns on client-side logging	8-7
xf_LogFile	Filename that logging output should be written to (rather than the screen)	8-7
xf_SessionRequestTimeout	Number of seconds <i>xfNetLink</i> should wait for acknowledgment from the connection monitor in <i>xfServerPlus</i>	8-8
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	8-8

Calling Synergy Routines from Java

Configuring the xfNetLink Java Properties File

Settings in the xfNetLink Java Properties File (continued)		
Setting	Description	See page
xf_DebugSessionConnectTimeout	Number of seconds xfNetLink should wait for acknowledgment from the logic server in xfServerPlus when running in debug mode	8-8
xf_SessionLingerTimeout	Number of seconds xfNetLink should wait for a return from a remote call	8-9
xf_SSLEncertFile	The path and filename of the keystore file to use for encryption	8-9
xf_SSLPassword	Password associated with the keystore file	8-9

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 distribution. 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 8-11](#).



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, or setting them in both ways. If both a property and its corresponding set method are specified, the set method take precedence. Use caution if you implement this approach; 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 set these time-outs, 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_DebugOutput	setxfLogging()
xf_LogFile	setxfLogfile()
xf_SessionRequestTimeout	N/A
xf_SessionConnectTimeout	N/A
xf_DebugSessionConnectTimeout	N/A
xf_SessionLingerTimeout	setxfCallTimeout()
xf_SSLCertFile	setSSLCertFile()
xf_SSLPassword	setSSLPassword()

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_DebugOutput  
xf_LogFile  
xf_SSLEntFile  
xf_SSLEntPassword
```

After an object is instantiated, if any of these items is set with the corresponding set method, 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. 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 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 details on logging see [“Using Client-Side Logging” on page 9-7](#).

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 exist, 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 4-2 on page 4-5](#)).

- 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 4-2](#)) 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 is used.

Connect session time-out

Once the initial socket communication is established ('A' in [figure 4-2](#)), the connection monitor signals the logic server to start a session. The connect session time-out ('B' in [figure 4-2](#)) 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 9-12](#)).

► 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
```

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 4-2](#)) 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 8-19. 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.

Specifying Encryption Options

These options are used to specify the certificate file and its password when you are using either master or slave encryption. If these values are present in the properties file, they are used instead of the defaults. You can override these values at runtime with the `setSSLCertFile()` and `setSSLPassword()` methods; see “[Method Reference](#)” on page 8-35. See “[Using Encryption](#)” on page 3-22 for more information on implementing encryption.

► To specify a certificate file and password

In the `xfNetLink` Java properties file, specify the full path and filename of the certificate file. The default value is the `cacerts` file located in the `java.home\lib\security` directory, where `java.home` is the JRE installation directory. (But note that we recommend *against* using this file, as it will be overwritten any time Java is updated.) For example,

```
xf_SSLCertFile = c:\\java\\jre6\\lib\\security\\myCertFile
```

Calling Synergy Routines from Java

Using Your JAR File

The default password is “changeit”. Use `xf_SSLPassword` to specify a different password. For example,

```
xf_SSLPassword = myFavPassword
```

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 8-1](#).



If you are using Java connection pooling, follow the steps in [“Using Your JAR File with Connection Pooling” on page 8-22](#) 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/Namespace 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 *xfNetLink* Java utility methods. The utility methods enable you to establish a connection with *xfServerPlus*, disconnect from *xfServerPlus*, and perform other utility functions, such as setting properties and running a debug session. See the method reference on [page 8-35](#) 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 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.

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. Use it when

- ▶ you are using an *xfNetLink* Java properties file named **xfNetLnk.ini** that resides 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, logging, and encryption 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`.

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 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 8-35](#) 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()` 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 *xfServerPlus* 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: xfNetLink Java Sample Code](#)” 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 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. (If you specified the `-ro` option when running **genjava**, fields that are flagged as read-only in Repository were generated as read-only properties and therefore have a get method but no set method.)



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.

You’ll also see a class named “Address” in the JAR file. This class represents the group within User. (Fields defined as struct data type in the repository are treated the same as group fields.) 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.

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 2 on page 8-15](#).

For additional information about structure classes, see [“Understanding the Generated Classes” on page 7-18](#). 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");
```

Calling Synergy Routines from Java

Using Your JAR File

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();
```

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 JAR file as enum type classes. There will be a separate class, named with the enumeration name, for each enumeration. Enumerations are supported only when **genjava** is run with the **-c 1.5** option (or the “Generate classes as version” option is set to 1.5 in Workbench).

The members in the repository enumeration are the values within the enum type. If you assigned numerical values to the members in repository, they are used for the integer equivalents assigned to the values in the generated classes. If not, the integer equivalents are assigned automatically starting with 0 and incrementing by 1. When you create a new instance of an enum type, it has a default value of the enumerator that has been assigned 0, if you do not explicitly assign a value. Consequently, when defining your enumeration in the repository, you should specify as the first member the value you would like to be the default.

Passing “out” and “in/out” parameters

Java doesn’t support a holder class for enumerated types, which means they can’t be passed as out or in/out parameters. We have included methods in your generated classes that give you a way to work around this limitation. `getIntHolderValue()` gets the integer value of an enumeration member and `getEnumeration()` associates that integer value with the enumeration member. See the examples below.

Examples

The following examples use an enumeration named `Color`, which has members `Green`, `Blue`, etc. (Note that the first letter of each member name is capitalized.) In the first example, the enum is passed as an in/out parameter of the `EnumTest1()` method, so we use the `getIntHolderValue()` and `getEnumeration()` methods. In the second example, the enum is a field within the `AcmeCustomer` structure class.

```
// Method call with an in/out enum parameter
AcmeCompanyComponent acme = new AcmeCompanyComponent();
acme.Color companyHat = acme.Color.Green;
acme.connect(host, port);
// use IntHolder to get integer value
IntHolder hatColor = companyHat.getIntHolderValue();
//pass IntHolder in method call
acme.EnumTest1(hatColor);
// assigne returned value to the companyHat
companyHat = acme.Color.getEnumeration(hatColor);
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(customer);
acme.disconnect();
```

Passing Binary Data

You can pass binary data, such as JPEG files, by using an `ArrayList`.



Binary fields in structures are converted to byte arrays (when **genjava** is run with the **-c 1.5** option or the “Generate classes as version” option is set to 1.5 in Workbench). 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.

In the MDU, define the parameter as a “Binary (handle)” data type. (If you attribute your code, see [example G on page 2-19](#) for instructions on defining a binary handle.) Your Synergy server routine must declare the argument that receives the data as a memory handle (**i4**). *xfServerPlus* will place the data in a

Calling Synergy Routines from Java

Using Your JAR File

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.

Passing “in” parameters

If the parameter is defined as “in” only in the SMC, in your Java code you will need to create a byte array and fill it with data, instantiate an `ArrayList` of byte arrays, and add the byte array as the first element of the list. Then, you can make the method call.



This example presumes that **genjava** was run with the **-c 1.5** option (or that the “Generate classes as version” option was set to 1.5 in Workbench), which generates generic `ArrayList`s. Consequently, it shows instantiating an `ArrayList` of byte arrays. When classes are generated as version 1.2, the `ArrayList`s are not generic, and the client code should instantiate a plain `ArrayList`.

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
.
.
//Create ArrayList of byte arrays
ArrayList<byte[]> al = new ArrayList<byte[]>();
al.add(inba); //Add byte[] as first element of ArrayList
JCWinstance.BinaryArrayMethod(al); //Call XFPL method
JCWinstance.disconnect();
```

Passing “out” parameters

For “out” only parameters, in your Java code you will need to instantiate an `ArrayList` of byte arrays, 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();
```

```
// Create ArrayList of byte arrays
ArrayList<byte[]> al = new ArrayList<byte[]>();
JCWinstance.BinaryArrayMethod(al); //Call XFPL method
byte[] rtnba = (byte[])al.get(0); //Extract byte[]
                                // from first element
JCWinstance.disconnect();
```

Passing “in/out” parameters

For “in/out” parameters, in your Java code you will need to create a byte array and fill it with data, instantiate an ArrayList of byte arrays, 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
.
.
//Create ArrayList of byte arrays
ArrayList<byte[]> al = new ArrayList<byte[]>();
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 4-2 on page 4-5](#).) 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

Calling Synergy Routines from Java

Using Your JAR File

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, and it will affect all objects that share that connection.

For example, to set the call time-out to 10 minutes, use

```
AppLogin appLog = new AppLogin();
appLog.setxfCallTimeout(600);
```

For information on specifying a call time-out value in the properties file, see [“Call time-out” on page 8-9](#). 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 8-29](#).

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 enabled (see [“Using Server-Side Logging” on page 3-30](#)) 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 login 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 Pooling

Java pooling enables you to create a “pool” of connections to *xyServerPlus* 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. However, if an exception has been thrown on the connection, it is discarded, regardless of how this option is set.

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 8-28](#) 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 8-1](#) 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 8-10](#) 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 `xfNetLink` 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.)

You can also pass the pooling properties file to use with the `getInstance()` method. See [SWPManager on page 8-42](#) for an example.

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 8-24.](#)) The `usePool()` method is one of the public utility methods that is included when you build your JAR file. (See the method reference on [page 8-35.](#))

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 8-12](#) 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. If your file is named **xfPool.properties** and placed in either the Java application directory or in the directory required by your web server and servlet container, you can use the `getInstance()` method that doesn't take any parameters; it will use this default file. However, you can choose to name the pooling properties file differently and place it elsewhere and then use one of the `getInstance()` methods that passes the properties file. See [SWPManager on page 8-42](#) for details on these methods.

There is a sample pooling properties file included in the Examples directory that is part of the *xfNetLink* Java distribution. You can copy the sample file to the correct location and modify it, or you can create your own text file with the necessary settings to use as a properties file.

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 8-30](#).

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. If you need to retrieve the pool ID, use the `getPoolName()` method. (See the method reference on [page 8-35](#).)

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 on [page 8-25](#) 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 8-30](#).

Settings in the Pooling Properties File		
Setting	Description	See page
minPool	(required) The minimum number of connections to be maintained in the pool.	8-27
maxPool	(required) The maximum number of connections to be maintained in the pool.	8-27

Calling Synergy Routines from Java

Implementing Pooling

Settings in the Pooling Properties File (continued)		
Setting	Description	See page
propertiesFile	(required) The path and filename of the <i>xfNetLink</i> Java properties file to use.	8-27
poolReturn	Boolean value that indicates whether the connection should be discarded or returned to the pool for reuse.	8-27
poolLogFile	Filename that logging information should be written to.	8-28
poolLogLevel	Level of logging desired. Possible values are none, error, and all. If <i>poolLogFile</i> is specified and <i>poolLogLevel</i> is not specified, error level logging will take place.	8-28
connectWaitTimeout	Number of seconds that the <i>getConnection()</i> method will wait for a connection from the pool.	8-29
initializationTimeout	Number of seconds that <i>xfNetLink</i> should 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 method specified with <i>initializationMethodID</i> .	8-29
initializationMethodID	Method ID of the Synergy method that will be called each time a new connection is added to the pool.	8-29
activationMethodID	Method ID of the Synergy method that will be called each time <i>getConnection()</i> is called.	8-29
deactivationMethodID	Method ID of the Synergy method that will be called each time a connection is freed.	8-29
poolableMethodID	Method ID of the Synergy method that will be called after the method specified by <i>deactivationMethodID</i> is called.	8-29
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).	8-29

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 *xf*ServerPlus 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 *xf*NetLink Java properties file to use

When using Java connection pooling, you must use an *xf*NetLink 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 *xf*NetLink Java properties file you will be using. This setting is required.

When you are using pooling, you can name the properties file anything you like and place it anywhere on the *xf*NetLink Java machine because the setting in the pooling properties file enables *xf*NetLink to find it. In addition, you can have more than one properties file (one per pool, if desired), even for a JSP application.

For example:

```
poolID.propertiesFile=c:\\tomcat\\conf\\poolxfnj.properties
```

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 8-21](#) 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.

Calling Synergy Routines from Java

Implementing Pooling

For example:

```
poolID.poolReturn=true
```

If poolReturn is not set, connections will be discarded.

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 8-29](#) and [“Pooling support methods” on page 8-33](#).

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, so you don’t need to specify the pool ID.

► 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 8-9 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 8-32 for more information on writing the methods.)

To use the pooling support methods, you must specify them by method ID—not method name—in the pooling properties file.

Calling Synergy Routines from Java

Implementing Pooling

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.properties
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.properties
```

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 *xy*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 *xy*NetLink Java properties file used by that pool. For more information about `resetPoolProperties()`, see [SWPManager on page 8-42](#).

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 *xy*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 [SWPManager on page 8-42](#).

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 8-33](#) for the syntax your routine should use, a description of the purpose of each method, and when each method is called.
2. Add the routines to the SMC, using either the Method Definition Utility or by attributing the Synergy code, running **dbl2xml**, and then importing the XML file. If you use the MDU, 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.) However, **dbl2xml** requires an interface name; you can use a different interface name than is used for your procedural methods to avoid including the pooling support methods when you build 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 8-29](#).

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 8-21](#) 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 [XFPL_REGCLEANUP on 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

Listed below are the public utility methods included in the procedural classes in your Synergy JAR file. All of these methods are available in all procedural classes with the exception of the four enumeration methods, which are included only when needed.

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 8-12](#).

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 9-10](#).

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 9-10](#).

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 8-14](#).

getConnect()

public Object getConnect() throws xfJCWException

Returns the already-established connection of the specified object. See [“Connect to xfServerPlus” on page 8-12](#).



The `getEnumeration()`, `getIntHolderValue()`, and `getIntHolder()` methods are included in a procedural class only when that class includes a method that has a parameter or return value that is an enumeration, or when a structure passed as a parameter includes an enumeration field.

getEnumeration()

```
public static getEnumeration(intHolder enumVal)
```

Associates an integer value with an enumeration member. This method is used in conjunction with `getIntHolderValue()` when you need to pass an enumeration parameter as in/out or out. See [“Using Enumerations” on page 8-16](#).

`enumVal`—the value that was obtained with `getIntHolderValue()`

getEnumeration()

```
public static getEnumeration(int enumVal)
```

Associates an integer value with an enumeration member. This method is called by the `getEnumeration()` method above; you do not need to call it directly.

`enumVal`—the value that was obtained with `getIntValue()`

getIntHolderValue()

```
public intHolder getIntHolderValue()
```

Returns an `intHolder` with the value of the enumeration member. This method is used in conjunction with `getEnumeration()` when you need to pass an enumeration parameter as in/out or out. See [“Using Enumerations” on page 8-16](#).

getIntValue()

```
public int getIntValue()
```

This method is called by `getIntHolderValue()`; you do not need to call it directly.

getPoolName()

```
public String getPoolName()
```

Returns the pool ID that was used in the call to the `usePool()` method by this object. If there is no pool (i.e., the `usePool()` method has not been called on this object), returns a null string.

getSSLCertFile()

```
public String getSSLCertFile()
```

Returns the path and filename of the certificate file stored by `setSSLCertFile()`. For more information on encryption, see [“Using Encryption” on page 3-22](#).

getSSLPassword()

```
public String getSSLPassword()
```

Returns the certificate file password stored by `setSSLPassword()`. For more information on encryption, see [“Using Encryption” on page 3-22](#).

getSynergyWebProxy()

```
public SynergyWebProxy getSynergyWebProxy()
```

Returns the internal instance of the `SynergyWebProxy` (i.e., the connection). If there is no connection, returns null.

getUserString()

```
public String getUserString()
```

Returns the string currently stored by `setUserString()`. See [“Writing to the xfServerPlus Log” on page 8-20](#).

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()`.

getxfPort()

```
public int getxfPort()
```

Returns the port number that was set with `setxfPort()`.

setSSLCertFile()

```
public void setSSLCertFile(String certFileName) throws  
xfJCWException
```

Sets the certificate file to be used for `xfNetLink` Java network encryption. For more information on encryption, see [“Using Encryption” on page 3-22](#).

`certFileName`—the path and filename of the certificate file to use

setSSLPassword()

public void setSSLPassword(String *password*) throws
xfJCWException

Sets the password associated with the certificate file. For more information on encryption, see [“Using Encryption” on page 3-22](#).

password—the password associated with the certificate file

setUserString()

public void setUserString(String *userString*) throws
xfJCWException

Passes a user string to the *xfServerPlus* log. See [“Writing to the xfServerPlus Log” on page 8-20](#).

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 8-19](#).

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 [“Using Client-Side Logging” on page 9-7](#).

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. Additional logging of methods and parameters 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 [“Using Client-Side Logging” on page 9-7](#).

logging—Boolean value to indicate that logging is on. Set this to true to activate logging.

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 8-12](#).

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 [SWPManager on page 8-42](#).

Class Reference

Synergex.util.SWPCConnect

java.lang.Object



public class SWPCConnect extends Object

The SWPCConnect 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, which actually makes the connection to *xfServerPlus*.

You do not need to call this class directly.

Constructors public SWPCConnect(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 SWPCConnect(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



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 Pooling” on page 8-21](#) and [“Implementing Pooling” on page 8-22](#).

Methods static synchronized public SWPManager getInstance() throws
xfPoolException

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 default pooling properties file (**xfPool.properties**). Once the pool has been created, subsequent calls to getInstance() get an instance of the pool.

static synchronized public SWPManager getInstance(String
propFile) throws xfPoolException

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 specified with *propFile*. You can use this method if you do not want to use the default pooling properties file. Once the pool has been created, subsequent calls to getInstance() get an instance of the pool.

propFile—the full path and filename of the pooling properties file

static synchronized public SWPManager getInstance(Properties
poolProp) throws xfPoolException

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 specified with *poolProp*. (You must first create the Properties object and load it.) This just offers another way to specify a non-default pooling properties file. Once the pool has been created, subsequent calls to getInstance() get an instance of the pool.

poolProp—a java.util.Properties object that represents the pooling properties file

```
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()`

Calling Synergy Routines from Java

SWPManager

Usage The `getInstance()` method of the `SWPManager` class instantiates the pool the first time it is called, then gets an instance of the pool on subsequent calls. If there are multiple pools defined in the pooling properties file, they are all created on the first call to `getInstance()`.

To use the default pooling properties file (which is named **xfPool.properties** and located either in the Java application directory or in the directory required by your web server and servlet container) call the `getInstance()` method that passes no parameters.

To specify a non-default pooling properties file, use either of the other two `getInstance()` methods.

For example, to specify the full path and filename of the pooling properties file as a `String`,

```
SWPManager poolMgr = SWPManager.getInstance("c:\\files\\  
myPool.properties");
```

Or, to use a `Properties` object to specify the pooling properties file,

```
Properties poolProps = new Properties();  
FileReader fr = new FileReader("c:\\files\\myPool.properties");  
poolProps.load(fr);  
SWPManager poolMgr = SWPManager.getInstance(poolProps);
```

See also [“Using Your JAR File with Connection Pooling” on page 8-22](#) 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 always catch this exception.

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 other *xfNetLink* Java exceptions (except for *xfPoolException*).

Read the error message to determine what the specific problem is and refer to the table on [page 9-1](#).

For more information on errors, see [“Handling Errors” on page 9-1](#).

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 Pooling” on page 8-21](#) and [“Implementing Pooling” on page 8-22](#). For more information on errors, see [“Handling Errors” on page 9-1](#).

Error Handling and Troubleshooting in *xfNetLink* Java

This chapter includes a table that lists the error that can occur 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 Errors

All *xfNetLink* Java exceptions are thrown as `xfJCWException`, which your code should catch. If you are using Java connection pooling, you should also catch `xfPoolException`. The error text associated with these exceptions will tell you what the specific problem is. Check the table below for the likely cause and possible solutions. 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 3-15](#).

<i>xfNetLink</i> Java Errors		
Error message	Cause	What to do
<numParms> is illegal parameter count. Range is 0 to 253	The method call has more parameters than permitted.	Correct your client code to pass fewer parameters.
14-day demo period expired	The 14-day evaluation period for your <i>xfServerPlus</i> license has expired.	Contact your Synergy/DE customer service representative to purchase a license.
Bad array, Index <number>	There is a problem with an element in an array.	The message will include additional information that can help you determine the nature of the problem.
Bad Packet: Type conflict prevented variable update after method call	The format of the returned data is incompatible with that of the sent data, or the wrong data type was used in a method call.	Check your client code to ensure you are using the correct data types.

Error Handling and Troubleshooting in xfNetLink Java

Handling Errors

xfNetLink Java Errors		
Error message	Cause	What to do
Bad packet. Improper delimiter after parameter	The packet is corrupted or badly formed.	Retry. The problem may be noise on the line or some other type of transmission error. If logging is turned on, you can examine the packet contents in the log.
Can't create a new connection	There was a problem creating a connection in the pool.	The information in the message can help you determine the source of the problem.
Can't declare an array with multiple SynTypes	You attempted to declare an array that contains more than one data type.	Correct your client code to use a single data type in the array.
Can't use array to store multiple SynTypes. Array declared as <declared type>. Element <name> is of type <deviant type>.	You attempted to store multiple data types in an array.	Correct your client code to use a single data type in the array.
Could not convert <value> to a SynByteArray. Value must be byte[]	Conversion to or from a byte array failed.	This error should not occur when using Java class wrappers, unless code in the generated Java classes has been altered or removed. If you know what was changed, you can attempt to correct the code, but the recommended solution is to regenerate the classes and rebuild the JAR file.
Could not convert <value> to a SynDec. Value must be integer	Conversion to or from a decimal data type failed.	
Could not convert <value> to a SynEnum. Value must be integer	Conversion to or from an enumeration failed.	
Could not convert <value> to a SynImpDec. Value must be integer or real	Conversion to or from an implied-decimal data type failed.	
Could not convert the first element of the ArrayList to a byte[]	Conversion to or from an ArrayList failed.	
Error parsing data to structure format	A structure parameter has an incorrect type.	Check your client code. If the repository was updated, you must also update the SMC, regenerate classes, and rebuild the JAR file.
Extended demo period expired	An extended evaluation period for your xfServerPlus license has expired.	Contact your Synergy/DE customer service representative to purchase a license.

Error Handling and Troubleshooting in xfNetLink Java

Handling Errors

xfNetLink Java Errors		
Error message	Cause	What to do
Failure occurred within remote Synergy routine	xfServerPlus cannot translate a method call into a Synergy routine call, or xfServerPlus has a problem (e.g., an error reading the SMC), or a routine that's being executed caused an error.	The information in the error can help you determine if the problem is in the Java code, the Synergy code, the SMC, or was generated by xfServerPlus.
Invalid connectWaitTimeout property setting	The connectWaitTimeout setting in the pooling properties file is invalid.	Correct the pooling properties file. See page 8-29 .
Invalid initializeTimeout property setting	The initializeTimeout setting in the pooling properties file is invalid.	Correct the pooling properties file. See page 8-29 .
Invalid maxPool property setting	The minPool setting in the pooling properties file is invalid.	Correct the pooling properties file. See page 8-27 .
Invalid Message Type in DTL	The packet is corrupted or badly formed.	Retry. The problem may be noise on the line or some other type of transmission error. If logging is turned on, you can examine the packet contents in the log.
Invalid minPool property setting	The minPool setting in the pooling properties file is invalid.	Correct the pooling properties file. See page 8-27 .
Invalid poolReturn property setting	The poolReturn setting in the pooling properties file is invalid.	Correct the pooling properties file. See page 8-27 .
Method Id Cannot Be Null	The method ID is missing from the method call.	This error should not occur when using Java class wrappers, unless code in the generated Java classes has been altered or removed. If you know what was changed, you can attempt to correct the code, but the recommended solution is to regenerate the classes and rebuild the JAR file.
Method Id not extracted because no delimiter found	The packet is corrupted or badly formed.	Retry. The problem may be noise on the line or some other type of transmission error. If logging is turned on, you can examine the packet contents in the log.

Error Handling and Troubleshooting in xfNetLink Java

Handling Errors

xfNetLink Java Errors		
Error message	Cause	What to do
Method Id not found in return result stream	The packet is corrupted or badly formed.	Retry. The problem may be noise on the line or some other type of transmission error. If logging is turned on, you can examine the packet contents in the log.
Missing required property setting for the maxPool connections	The maxPool setting is missing from the pooling properties file.	Add the setting to the pooling properties file. See page 8-27 .
Missing required property setting for the minPool connections	The minPool setting is missing from the pooling properties file.	Add the setting to the pooling properties file. See page 8-27 .
Missing required property setting for the pool ID	The setting that identifies the named pool is missing from the pooling property file.	Define the pool ID setting (e.g., myPool.pool) in the pooling properties file. See “Setting Up a Pooling Properties File” on page 8-24 .
Missing required property setting for the xfNetLink Java properties file	The propertiesFile setting, which specifies the location of the xfNetLink Java properties file, is missing from the pooling properties file.	Add the setting to the pooling properties file. See page 8-27 .
No hostname found in xfNetLink properties file	The xfServerPlus machine name is missing from the xfNetLink Java properties file	Correct the xfNetLink Java properties file or specify the host name with setxfHost(). See “Specifying the Host Name and Port Number” on page 8-6 .
No pool description specified	There are one or more settings in the pooling properties file that are missing the pool ID.	Correct the pooling properties file. Each setting must be preceded by the pool ID (e.g., myPool.maxpool). See “Setting Up a Pooling Properties File” on page 8-24 .
No port number found in xfNetLink properties file	The port number is missing from the xfNetLink Java properties file.	Correct the xfNetLink Java properties file or specify the port number with setxfPort(). See “Specifying the Host Name and Port Number” on page 8-6 .
Number of licensed concurrent connections exceeded	xfServerPlus has exceeded the number of available licenses.	Contact your Synergy/DE customer service representative to purchase additional licenses.

Error Handling and Troubleshooting in xfNetLink Java

Handling Errors

xfNetLink Java Errors		
Error message	Cause	What to do
Port number in xfNetLink properties file must be an integer	There is an invalid value for port number in the xfNetLink Java properties file.	Correct the xfNetLink Java properties file. See “Specifying the Host Name and Port Number” on page 8-6 .
Received bad packet. Length or num-elements inconsistent with amount expected	The method call contains more or fewer parameters than expected, or the size of a parameter does not match the expected size.	Check your client code to verify you are passing the correct parameters in the call. Also, make sure your Synergy routines, repository, SMC, and JAR file are all in sync.
Remote Synergy execution aborted	There was a fatal, untrappable error during the execution of a Synergy routine.	For additional information, check the xfServerPlus log or the event log (Windows), syslog (UNIX), or operator console (OpenVMS).
Server version <i>n</i> is incompatible with the required version <i>n</i>	The version of xfServerPlus is not compatible with the version of xfNetLink for the current operation.	Upgrade xfServerPlus to the current version. The message includes the protocol version; see the table on page 3-40 for how it maps to the software version.
Severe system error	A serious error, such as inadequate memory, occurred while xfServerPlus was starting up.	Retry. If the problem persists, save the error log and contact Synergy/DE Developer Support.
The minPool cannot be less than one!	The minimum pool size must be at least one.	Increase the value of the minPool setting in the pooling properties file. See page 8-27 .
The minPool is greater than maxPool setting	The minPool setting cannot be greater than the maxPool setting.	Adjust the minPool and maxPool settings in the pooling properties file. See page 8-27 .
xfServer denied access to remote execution	xfServerPlus cannot start a session because it is not licensed, or the number of licenses has been exceeded, or it cannot access the license database, or there is some problem with the account used to start xfServerPlus, such as invalid username or password.	Check your xfServerPlus account setup and licensing.

Error Handling and Troubleshooting in xfNetLink Java

Handling Errors

xfNetLink Java Errors		
Error message	Cause	What to do
xfServerPlus not supported by this xfServer	The version of <i>xfServer</i> that is running does not support <i>xfServerPlus</i> .	Upgrade <i>xfServerPlus</i> to the current version.
xfServerPlus session did not connect	<i>xfServerPlus</i> was started without the remote execution option.	Restart rsynd with the -w option.
Zero length packet received, host terminated connection	Slow startup caused a time out.	Retry. See “Connect session time-out” on page 8-8.

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 Java client-side logging is not as extensive as the server-side logging available with *xfServerPlus*. See [“Using Server-Side Logging” on page 3-30](#) for more information. You may also want to run the test program included with *xfServerPlus*; see [“Testing xfServerPlus” on page 3-14](#).

Using 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 if encryption is enabled, the log displays a string of 10 asterisks instead of the packet data for encrypted methods.

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 8-7](#) 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. The log files will contain data only from the class from which they are called. See the method reference on [page 8-35](#) 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: July 31, 2012 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: July 31, 2012 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 2-38](#).

Error Handling and Troubleshooting in xfNetLink Java

Troubleshooting Techniques

► To run the xfNLJTest program

1. Make sure *xfServerPlus* has been started on the server machine. (See “[Running xfServerPlus](#)” on page 3-2.)
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*, which may be helpful in troubleshooting.



(Windows) If you see the message “xfNLJTest.txt (Access is denied)”, it means that the user account under which you are logged on does not have write permission for the \Program Files directory, and so the logfile cannot be written. Just copy the **xfNLJTest.class** file to a writable location and run the test from there.

You may receive errors in the form of Java exceptions, which are also written to **xfNLJtst.txt**. The error message text should help you determine what needs to be done to correct the problem. Check the “[xfNetLink Java Errors](#)” table on page 9-1. You may want to also run the *xfServerPlus* test program, **xfsplttst**. It can help you determine whether the problem is on the *xfNetLink* side or the *xfServerPlus* side. See “[Testing xfServerPlus](#)” on page 3-14 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 Developer 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 3-45 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). (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*.

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());
```

Error Handling and Troubleshooting in xfNetLink Java

Troubleshooting Techniques



If you are debugging through a firewall, you may need to specify a port number range, and then open that range of ports on your firewall. To do this, call the `setxfMinport()` and `setxfMaxport()` methods in your client application prior to calling `debugInit()`, and pass as parameters the port numbers for the range. Valid values are greater than 1024, with `maxport` greater than `minport`.

Note that these methods are deprecated because they are not used by an ordinary connection; they are used only when running in debug mode. They are still included in your JAR file, but no longer documented in this manual.

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 8-8](#)). 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 Traditional Synergy Programs](#)” chapter in *Synergy 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 10.1.1    ***
BREAK AT 152 IN XFPL (LAUNCHER.DBL;6) ON ENTRY
%DBG-E-Could not open source file "LAUNCHER.DBL;1"
Dbldb>
*****
```

Error Handling and Troubleshooting in xfNetLink Java

Troubleshooting Techniques



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 Traditional Synergy Programs” chapter of *Synergy 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 Traditional Synergy Programs” chapter of *Synergy Tools*.

Part IV: *xf*NetLink .NET Edition

Creating Synergy .NET Assemblies

This chapter gives an overview of the tasks you must perform to set up and use *xfNetLink* .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 *xfNetLink* .NET classes internally to connect to *xfServerPlus*. The assembly that you build can be used in any .NET client application to remotely access your Synergy business logic on the *xfServerPlus* machine.

System Requirements

To build a distributed system with *xfNetLink* .NET and *xfServerPlus*, you'll need the .NET Framework version 2 or higher. The .NET Framework is installed with Visual Studio, or you can download it from the Microsoft website. If you do the latter, be sure to get the *full* version, which is intended for developers. This was referred to as an SDK for version 2, but later versions use different terminology, such as “full package” or “full profile”.

We recommend that you also install Microsoft Visual Studio version 2008 or higher, but Visual Studio is not required to build a Synergy assembly. If you do not have Visual Studio, the requirements depend on the version of the .NET Framework you are using:

- Framework 2 or 3.5: You can create a Synergy assembly without installing additional software. When you build the assembly, use **sdvars.bat** to set up the .NET environment.
- Framework 4 or 4.5: You will need to install either Visual C# 2010 Express or Visual Studio Express 2012, along with the Windows SDK 7.1. These applications will provide the tools necessary to build a Synergy assembly. When you build the assembly, use **vsvars32.bat** to set up the .NET environment.

You'll also need to be familiar with some .NET Framework concepts. 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 10-1 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 *xfNetLink .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 *xfServerPlus*, 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 *xfServerPlus* license.

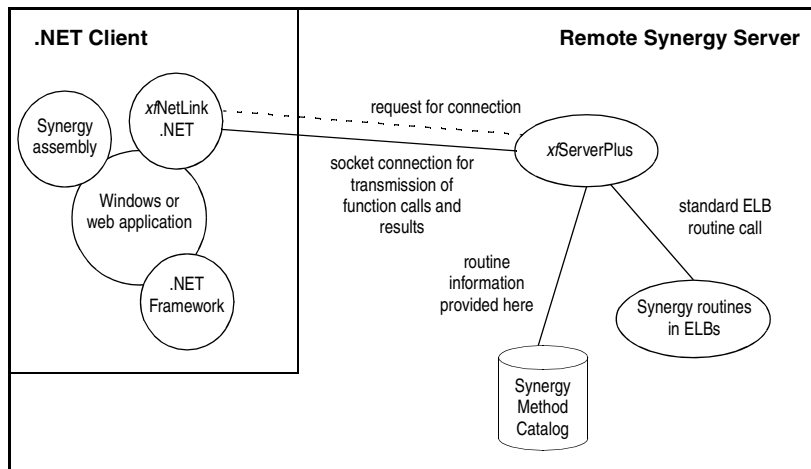


Figure 10-1. Accessing Synergy from .NET.

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.
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 DBL 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. You may also need to set other options in the **xfpl.ini** file; see [“Appendix A: Configuration Settings”](#) 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 3-2](#).
7. Generate the C# classes:
 - ▶ If you’re using Workbench, create a Synergy/DE .NET Component project. Specify the component information: a name and location for the assembly, the location of the SMC and repository files, the namespace and key file, the interfaces you want to include, and (if desired) alternate names for those interfaces. Then use the menu option to generate the C# classes. See [“Creating a Synergy/DE .NET Component Project” on page 10-6](#) and [“Generating C# Classes” on page 10-11](#).
 - ▶ If you’re using the command line, run the **genxml.dbr** utility to create an XML file, and then run **gencs.exe** to generate the C# classes. See [“Creating an Assembly from the Command Line” on page 10-14](#).
8. (optional) Modify the generated C# source files and the metadata for the assembly as necessary. See [“Editing the Generated Files” on page 10-26](#).
9. (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 & System Options* manual.

10. Build the assembly:
 - ▶ If you're using Workbench, use the menu option to compile the C# classes and create an assembly. See [“Building the Assembly” on page 10-12](#).
 - ▶ If you're using the command line, 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 10-21](#).
11. (optional) Generate the API documentation for the methods in your assembly using a third-party utility. See [“Generating API Documentation” on page 10-27](#).
12. 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 11-1](#) and [“Using Your Synergy .NET Assembly” on page 11-9](#).

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.

If you want to include the Original property or INotifyPropertyChanged class in structure classes, you must use the command line, as these options are not available in Workbench.

See [“Creating an Assembly from the Command Line” on page 10-14.](#)

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*.
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

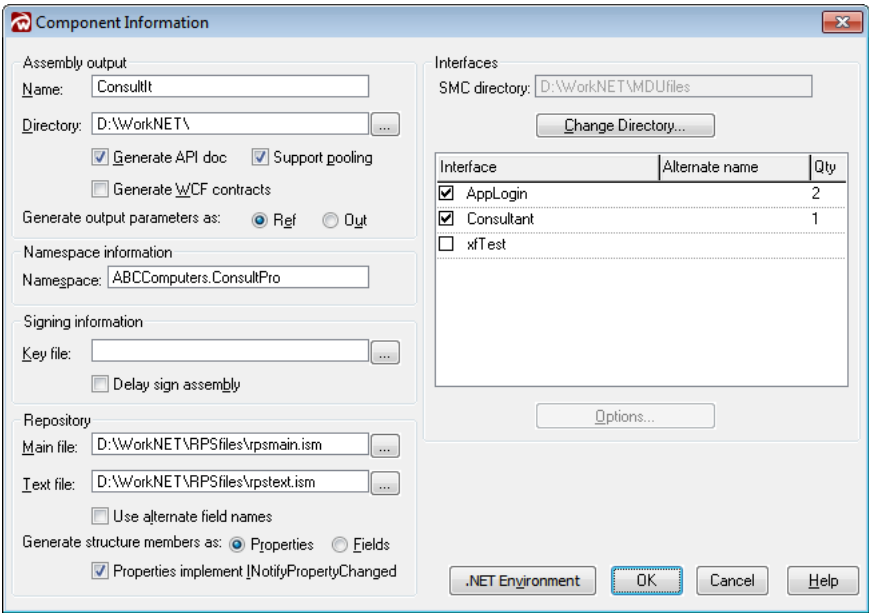


Figure 10-2. The Component Information dialog box for a Synergy/DE .NET Component project.

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 doc. 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 10-27](#) for more information.

Support pooling. Indicate whether you want to enable the classes in this assembly to be pooled.



Selecting the “Support 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 11-25](#) and [“Implementing Pooling” on page 11-27](#) for more information on pooling.

Creating Synergy .NET Assemblies

Creating an Assembly in Workbench

Generate WCF contracts. Select this option if you want the assembly to use WCF contracts, such as for a web service. The generated classes will include code that makes the assembly hostable. In addition, this option changes the way ArrayList and structure collection parameters are handled: rather than being generated as ArrayLists, they are generated as List<T> parameters (where T is the data type of the elements). Consequently, in your client code, you must use a List<T> class instead of an ArrayList class for a parameter that is defined as ArrayList or structure collection in the SMC.

Generate output parameters as. Indicate whether you want “out” parameters (that is, parameters with a direction of “out” in the SMC) to be generated as “ref” or “out” types in the C# classes. Array parameters are always generated as “ref”, regardless of how this option is set. (Prior to 9.5, all “out” parameters were generated as “ref”.)

Namespace. 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. 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. *x/fNetLink .NET* will verify that the file exists. See [“Using Your Own Key File” on page 10-25](#) for more information.

If you do not supply a key file name, *x/fNetLink .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 10-25](#) for more information.

Repository main file. 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. 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.

Properties implement INotifyPropertyChanged. Select this option to include the INotifyPropertyChanged class in generated structure classes. This enables you to use the PropertyChanged event on structure fields (properties) bound to controls.

SMC directory. This field displays the path for the Synergy Method Catalog that this assembly uses. The default is XFPL_SMCPATH; if it is not set, 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

Creating Synergy .NET Assemblies

Creating an Assembly in Workbench

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. 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 11-12](#) 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 nn COMNTOOLS%vsvars32.bat (where nn 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, depending on your needs. See [“System Requirements” on page 10-1](#) for details on the supported versions of the .NET Framework and Visual Studio requirements.

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. 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 2-53](#) for more information).
 - ▶ Create a C# source file for each interface you selected in the Component Information dialog box and for each structure, group, and enumeration within the selected interfaces. There will also be a C# interface file generated for each selected interface. See [“Understanding the Generated Classes” on page 10-23](#) 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 10-26](#).
 - ▶ 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 10-3 on page 10-13](#).

Creating Synergy .NET Assemblies

Creating an Assembly in Workbench

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 10-26](#).

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 & 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 doc” flag was set in the Component Information dialog box when you created the classes. See [“Generating API Documentation” on page 10-27](#) for more information on what to do with this XML file.

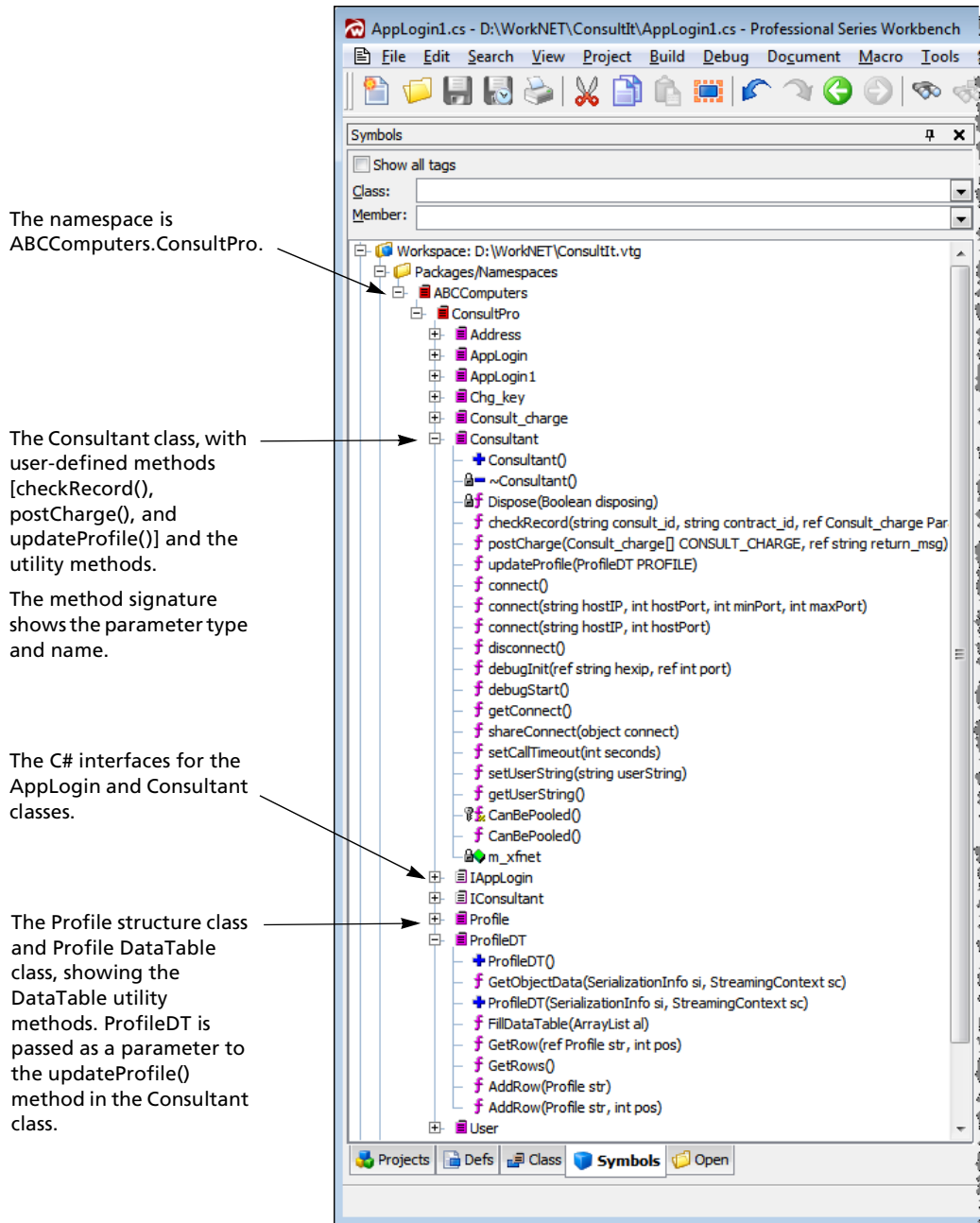


Figure 10-3. Viewing the generated C# classes and C# interfaces in Workbench.

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 an XML file. See [“The genxml Utility”](#) below.
2. Use the **genscs** utility to generate the C# classes from the XML file. See [“The genscs Utility”](#) on page 10-17.
3. (optional) Edit the source files. See [“Editing the Generated Files”](#) on page 10-26.
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 10-21.
5. (optional) Complete the API documentation using a third-party utility. See [“Generating API Documentation”](#) on page 10-27.

The genxml Utility

The **genxml** utility creates an 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 2-41.) Failure to do so can cause errors at runtime because the structure information in the component, which was pulled from the repository, 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 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 10-16](#).



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 .NET Assemblies

Creating an Assembly 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 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 an 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
```

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, group, and enumeration 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 10-26](#). 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] [-r] [-nr] [-p] [-i interfaceName:count] -w [-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.

Creating Synergy .NET Assemblies

Creating an Assembly from the Command Line

-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 10-25](#) for more information.

If not passed, *xfNetLink .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 10-25](#) for more information.

-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 10-27](#) 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, Changed property, or the INotifyPropertyChanged event, 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: Data Type Mapping”](#) for more information on data type conversion. Note that you should use the procedure described in [“Passing Binary Data”](#) on page 11-22, rather than a binary structure field, to pass most binary data.

-r

(optional) Include the Original property in structure classes. This property is used to store a copy of a structure within the object. By default it is not included. This option is not available when creating an assembly from Workbench. See [“Using the Original property”](#) on page 11-16 for details.

-nr

(optional) Generate output parameters (that is, parameters marked as “out” in the SMC) as “out” types in the C# code, with the exception of arrays, which are always generated as “ref”. If not passed, output parameters are generated as “ref” types in the C# code. (Prior to 9.5.1, all “out” parameters were generated as “ref” by default.)

-p

(optional) Include the INotifyPropertyChanged class in generated structure classes. This enables you to use the PropertyChanged event on structure fields (properties) bound to controls. This option cannot be used with **-g**; it is valid only when you generate properties.

Creating Synergy .NET Assemblies

Creating an Assembly from the Command Line

-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 10-20](#).) 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 11-12](#) for more information on using this feature.)

-W

Generate WCF contracts. Use this option if you want the assembly to use WCF contracts, such as for a web service. The generated classes will include code that makes the assembly hostable. In addition, this option changes the way ArrayList and structure collection parameters are handled: rather than being generated as ArrayLists, they are generated as List<T> parameters (where T is the data type of the elements). Consequently, in your client code, you must use a List<T> class instead of an ArrayList class for a parameter that is defined as ArrayList or structure collection in the SMC.

-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 10-23](#) and “[Editing the Generated Files](#)” on [page 10-26](#).

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 **SYNCSOPT** environment variable to any valid C# compiler command before running the batch file. See **SYNCSOPT** in the “*Environment Variables*” chapter of the *Environment Variables & System Options* manual for more information.

1. Run **vsvars32.bat** to set the environment variables used by .NET. This batch file is installed with Visual Studio (or with Windows SDK 7.1) and should be run whenever you run commands from the command line. Depending on your setup, you may want to run **sdvars.bat** instead; it is installed with versions 2 and 3 of

Creating Synergy .NET Assemblies

Creating an Assembly from the Command Line

the .NET Framework (but not version 4). (See “[System Requirements](#)” on [page 10-1](#) for details on supported versions of the .NET Framework and Visual Studio requirements.)



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 11-25](#) and “[Implementing Pooling](#)” on [page 11-27](#) 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, group, and enumeration within the selected interfaces. These generated classes use the *xfNetLink* .NET classes internally to connect with *xfServerPlus* and pass data.



See [“Appendix B: Data Type Mapping”](#) 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()  
getConnect()  
shareConnect()  
debugInit()  
debugStart()  
setUserString()  
getUserString()  
setCallTimeout()
```

The utility methods enable you to establish a connection with *xfServerPlus* 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 11-9 and the method reference on [page 11-37](#) for more information. Note that although they are included in the C# classes, `getConnect()` 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 11-12 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 11-13](#). For more information about the utility methods, see the method reference on [page 11-37](#).

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 11-17](#). For more information about the utility methods, see the method reference on [page 11-37](#).

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 11-21.

Custom Attributes

xfNetLink .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, *xfNetLink* .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.

Creating Synergy .NET Assemblies

Editing the Generated Files

► 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.
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 10-6](#) 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 10-17](#).

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 10-27](#)).



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 *xfNetLink* .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 2-20](#).

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 doc” 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_name*API.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, Sandcastle, is required to create the API documentation using the XML file (*assembly_nameAPI.xml*) produced when you built your assembly. Sandcastle is available on CodePlex at <http://shfb.codeplex.com>.



Microsoft originally developed Sandcastle and downloads can still be found on the Microsoft website, but they are not the most recent. We recommend you go to the CodePlex site instead to get information about the product and to download the latest version.

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 *xfServerPlus* 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 3-47](#).

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.
2. Install *xfNetLink* .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 *xfNetLink* .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 11-5](#).) See “[Using an Application Configuration File](#)” on [page 11-3](#) 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 11-25](#) and “[Implementing Pooling](#)” on [page 11-27](#).

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 from the Microsoft website.
2. Install *xfNetLink* .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 *xfNetLink* .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 11-27](#).

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 *xfNetLink* .NET distribution includes the *xfNetLink* .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 *xfNetLink* .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:\xfnl\net.log
logging	none (i.e., logging is off by default)
pool return	false
port	2356
single log file	off

xfNetLink .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 11-10](#) and the method reference on [page 11-37](#) 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 find and use the file.

- ▶ If you are using pooling, the file must be named **dllhost.exe.config** and reside in the `%windir%\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 `xfNetLink .NET` Configuration utility (**synnetcfg.exe**) enables you to create and edit application configuration files for your `xfNetLink .NET` applications. The config files created by the `xfNetLink .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 *xfNetLink* .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 *xfNetLink* .NET Configuration utility to add the *xfNetLink* .NET information to the file.

► To start the utility

From the Windows Control Panel, select Synergy Control Panel > *xfNetLink* .NET Configuration.

You can also run the utility from the menu option in Workbench. Select Synergy/DE > Utilities > *xfNetLink* .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.

► 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 11-4.

Calling Synergy Routines from .NET

Using an Application Configuration File

► 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 *xfNetLink* .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 11-6](#).
5. Click OK. Repeat this procedure for each setting you want to add.

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 4-2 on page 4-5 .) 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 4-2 on page 4-5).
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 Initialize() method. The default is 30 seconds. After the Initialize() method is called, the call time-out is reset to 30 minutes. See "Setting a Call Time-Out" on page 11-23 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.

Class Settings (continued)	
Key	Set value to
logging	<p>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 12-7 for more information and a sample log.</p> <ul style="list-style-type: none"> ▶ 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., connect(), disconnect()). 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.
pool return	<p>(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 11-25 for more information.</p>
port	<p>The port that xfServerPlus is listening on. The port number must be in the range 1024 through 65534.</p>
single log file	<p>“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 12-7 for additional information.</p>

Calling Synergy Routines from .NET

Using an Application Configuration File

► 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.

► 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 11-34](#).

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 *xfNetLink* .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 *xfNetLink* .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 11-37](#) 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 *xfServerPlus* 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.

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 webpage, 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 10-10](#) 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 10-17](#) and the examples on [page 10-20](#) for more information.

- ▶ Set the host name and port number for each copy of the class using the **xfNetLink** .NET Configuration utility. See “[Setting options in the xfNetLink .NET Configuration utility](#)” on [page 11-27](#).
- ▶ 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 11-34](#).
- ▶ 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 11-27](#) 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: Data Type Mapping](#)” 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
```

Calling Synergy Routines from .NET

Using Your Synergy .NET Assembly

```
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. You'll also see a class named "Address" in the assembly. This class represents the Address group within the User structure. (Fields defined as struct data type in the repository are treated the same as group fields.) 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.



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 11-37](#).

Calling Synergy Routines from .NET

Using Your Synergy .NET Assembly

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 structure classes when you use the `gens -r` option; 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 which (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
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" on page 11-19](#) and ["Examples" on page 11-20](#).)

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.

Rows and columns

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.)

Characteristics of the structure fields, such as non-nullable and read-only, are carried over to the DataTable. For example, for a non-nullable field, the data column's AllowDBNull property is set to false, and for a read-only field, the IsReadOnly property is set to true. The default value for a DataTable column is set to the default value for the data type of the column. The default applies when no data is present, such as when you create a new row.

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

Calling Synergy Routines from .NET

Using Your Synergy .NET Assembly

See the method reference on [page 11-37](#) 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 has been assigned 0, if you do not explicitly assign a value. Consequently, when defining your enumeration in the repository, you should specify as the first member, the value you would like to be the default.

The following examples use an enumeration named **Color**, which has members **Green**, **Blue**, etc. (Note that the first letter of each member name is capitalized.) 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();
```

Calling Synergy Routines from .NET

Using Your Synergy .NET Assembly

```
// 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 10-19](#).

In the MDU, the parameter must be defined as a “Binary (handle)” data type. (If you attribute your code, see [example G on page 2-19](#) 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**). *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.

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 4-2 on page 4-5](#)). This time-out is measured for *each* send–receive request between the client and *xfServerPlus*. The default value is 1800 seconds (30 minutes).

Calling Synergy Routines from .NET

Using Your Synergy .NET Assembly

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 11-6](#).

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 3-30) 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 *xfServerPlus* 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 *xfNetLink* .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 *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 11-31.](#))

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, 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 *xfNetLink* .NET Configuration utility, select the pool return key and select the “Return objects to pool” checkbox (see [page 11-7](#)). You can also determine at runtime if an object should be reused using the `CanBePooled()` method (see [page 11-33](#)).

Tips for Creating Poolable Objects

To enable the classes in an assembly to be pooled, select the “Support 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 *xfServerPlus* 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 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 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 11-31](#).
2. Create your assembly. If you wrote pooling support methods, include the interface that contains them in the assembly. Select the “Support 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 10-12](#) for building from Workbench or [page 10-21](#) for building from the command line.
3. Write your client-side code. See [“Writing Code That Uses Pooled Objects” on page 11-34](#).
4. (optional) Set the desired options in the *xfNetLink* .NET Configuration utility. See [“Creating and Editing Configuration Files” on page 11-4](#).
5. Run `regsvcs.exe` to create a pool. See [“Creating a pool” on page 11-28](#).
6. Configure the pool in Component Services. See [“Configuring the pool” on page 11-28](#).
7. Start the pool, and then run your application. See [“Starting the pool” on page 11-31](#).

Setting Up the Client Machine

To set up the client machine, you may want to set some options in the *xfNetLink* .NET Configuration utility. Then, you'll create, configure, and start the pool.

Setting options in the *xfNetLink* .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 11-3](#).) To do so, run the *xfNetLink* .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 11-4](#) 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 11-31](#).

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 11-1 on page 11-29](#).)
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 11-1](#)) and select Action > Properties.
5. In the Properties dialog box go to the Activation tab. (See [figure 11-2 on page 11-30](#).)

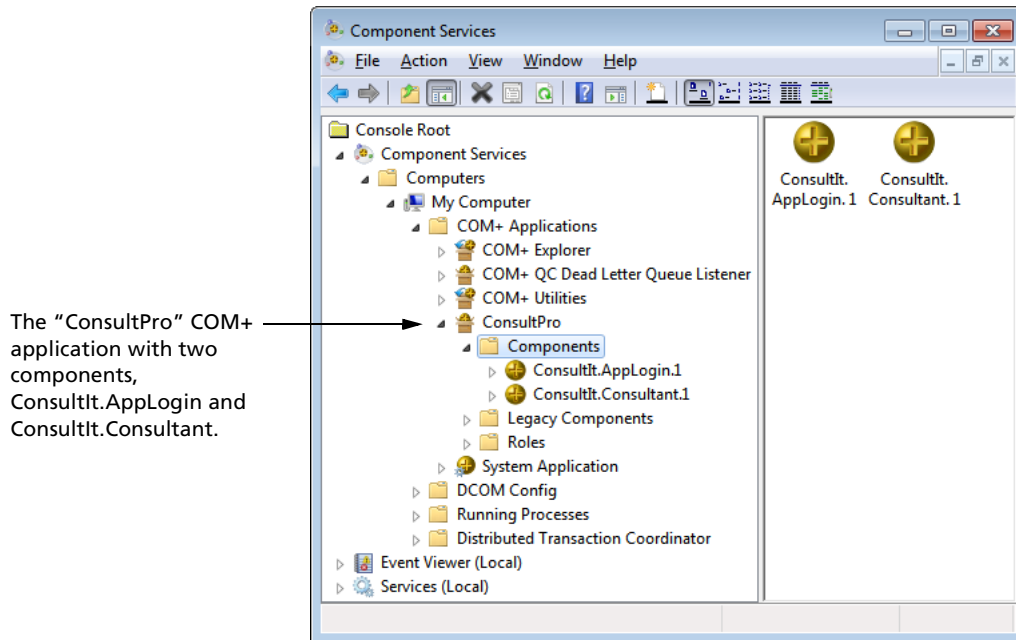


Figure 11-1. The Component Services window, showing a COM+ application with components.

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 *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 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.

Calling Synergy Routines from .NET

Implementing Pooling

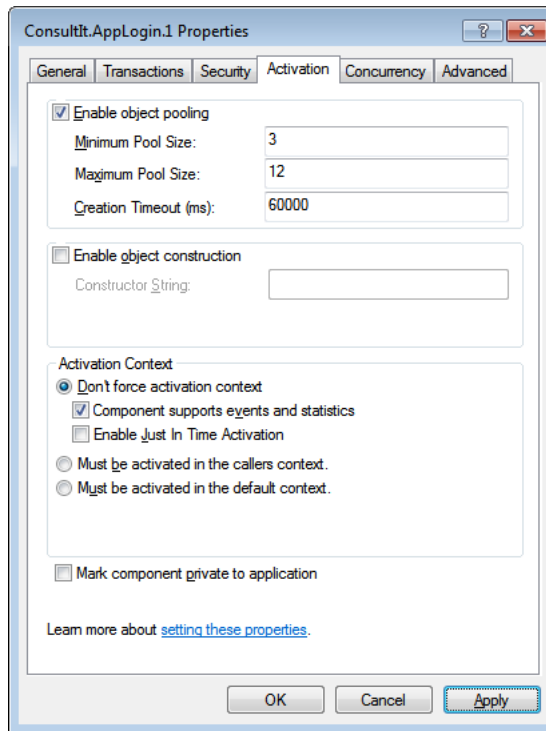


Figure 11-2. Setting properties for a pooled component.

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 11-1](#)). 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 *xfNetLink* .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

xfNetLink .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.

Calling Synergy Routines from .NET

Implementing Pooling

► 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. Add the routine to the SMC, using either the MDU or by attributing the Synergy code, running `dbl2xml`, and then importing the XML file. Note that because these methods are called automatically, they *must* be named with one of the following: Initialize, Activate, Deactivate, CanBePooled, Cleanup.
 - In the MDU, enter one of these names in the Method name field.
 - If you are attributing your code, you may need to use the name property of the `xfMethod` attribute to specify one of these names.



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 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 11-7](#)).

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 x/ServerPlus to lose socket communication with the client, x/ServerPlus 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 x/ServerPlus 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 11-9](#). Because the pooled objects already have a connection to x/ServerPlus, you do not need to call the connect() method. The disconnection from x/ServerPlus 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 x/NetLink .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.

The following code shows the same example written in VB.NET.

```
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 11-12](#) 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,

Calling Synergy Routines from .NET

Implementing Pooling

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

These methods 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 11-10](#).

```
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.

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 12-10](#).

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 12-10](#).

disconnect()

```
public void disconnect()
```

Sends a message to close the *xfServerPlus* connection. See [“Disconnect from xfServerPlus” on page 11-12](#).

getConnect()

```
public object getConnect()
```

Gets the already-established connection of the specified object. See [“Connect to xfServerPlus” on page 11-10](#). 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 11-24](#).

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 11-23](#).

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 11-24](#).

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 11-10](#). 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 11-15](#) for more information and examples.

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 Data Tables” on page 11-17](#) 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 the specified location in the DataTable; 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

Calling Synergy Routines from .NET

Method Reference

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 3-15](#).



When you get an error, the log file can provide information about what went wrong. See [“Using Client-Side Logging” on page 12-7](#). All errors are also written to the Windows application event log, regardless of whether client-side logging is turned on.

<i>xfNetLink .NET</i> 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	<i>xfNetLink .NET</i> timed out because of very slow start up.	Increase the connect time-out value. See “Creating and Editing Configuration Files” on page 11-4 .
Could not create new process	(Win, UNIX) Could not launch dfs process.	Ensure that dfs.exe exists on the <i>xfServerPlus</i> machine.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors (continued)		
Error message	Cause	What to do
DBLDIR not set	The DBLDIR environment variable is not set on the server.	Set DBLDIR on the <i>xfServerPlus</i> 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. <i>xfNetLink .NET</i> supports only the DateTime data type for dates.	Correct your client code to use a DateTime data type. See " xfNetLink .NET " in " Appendix B: Data Type Mapping ".
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 <i>xfNetLink .NET</i> Configuration utility to verify that the host and port are set correctly. See " Creating and Editing Configuration Files " on page 11-4. If you are not using a config file, verify that <i>xfServerPlus</i> is running as localhost on port 2356.
Incompatible client version	The <i>xfNetLink .NET</i> version is incompatible with the <i>xfServerPlus</i> 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 <i>xfServerPlus</i> 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.
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.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors (continued)		
Error message	Cause	What to do
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 2-41.
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 3-15 .
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 11-4.
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 3-2.
Method information must be passed to callMethod	Attributes for the method being called are not being passed to <i>xfNetLink</i> . (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.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors (continued)		
Error message	Cause	What to do
Missing class attribute for <i>className</i>	An attribute for <i>className</i> 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 gengcs) and rebuild the assembly.
Missing field attribute for <i>fieldName</i>	An attribute for <i>fieldName</i> is missing from a structure class. This attribute contains the Synergy data type and length of the field.	
Missing method attribute for <i>methodName</i>	An attribute for <i>methodName</i> is missing. This attribute contains the name of the method in the SMC to call.	
Missing parameter attribute for <i>parameterName</i>	An attribute for <i>parameterName</i> is missing. This attribute contains the Synergy data type and length of the parameter.	
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 xfServerPlus machine.	Install a Runtime license on the xfServerPlus machine.
Number of fields does not match	The number of fields returned by xfServerPlus 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 xfServerPlus 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 gengcs) 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.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors (continued)		
Error message	Cause	What to do
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 gensc) and rebuild the assembly.
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: Data Type Mapping ”. 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: Data Type Mapping ”.
<i>xfServerPlus</i> license count exceeded	<i>xfServerPlus</i> has exceeded the number of available licenses.	Contact your Synergy/DE customer service representative to purchase additional licenses.

Error Handling and Troubleshooting in xfNetLink .NET

Handling Errors

xfNetLink .NET Errors (continued)		
Error message	Cause	What to do
<i>xfServerPlus</i> license expired	The 14-day demo period or an extended demo period has expired.	Contact your Synergy/DE customer service representative to purchase a license.
<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 the port specified in the config file or passed by the connect() method.
<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* .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 3-30](#)) and run the test program included with *xfServerPlus* (see [“Testing xfServerPlus” on page 3-14](#)).

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 *xfNetLink* .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 *xfNetLink* .NET Configuration utility to turn on the “single log file” option. (See [“Creating and Editing Configuration Files” on page 11-4](#) 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 *xfNetLink* .NET Configuration utility; see the table on [page 11-7](#) 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: 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 = 9.5.3.1
*****
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***&*****06**3*****
00046X:received launch response packet
00046X:packet length=52
00046X:4***&***,*****Y*=*****
00046X:received acknowledgement packet
00046X:packet length=53
00046X:5*****31-JUL-2012 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: 31-JUL-2012 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: 31-JUL-2012 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 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 2-38](#).

There is also a test program for *xfServerPlus*; see [“Testing xfServerPlus” on page 3-14](#).

► 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 11-3](#)) will be used unless you edit the application configuration file for the test program. If necessary, start the *xfNetLink .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 *xfNetLink .NET* Configuration utility, see [“Creating and Editing Configuration Files” on page 11-4](#).
 - Save the configuration file and close the utility.



If you get a permissions error on saving the config file, it means that the user account under which you are logged on does not have write permission for the \Program Files directory. Just copy the entire \Examples directory to a writable location and run the test from there.

Error Handling and Troubleshooting in xfNetLink .NET

Troubleshooting Techniques

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 12-1](#). 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 3-45](#) 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);
```



If you are debugging through a firewall, you may need to specify a port number range, and then open that range of ports on your firewall. To do this, manually edit the application configuration file to include the following lines within the `<xfnlNet>` section of the file:

```
<add key="minport" value="####" />
<add key="maxport" value="####" />
```

where `####` is a port number greater than 1024, with `maxport` greater than `minport`.

Note that these settings are not used by an ordinary connection; they are used only when running in debug mode and, consequently, have been removed from the *xfNetLink .NET* Configuration utility.

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 *xfNetLink .NET* Configuration utility. See ["Creating and Editing Configuration Files"](#) on page 11-4.

Error Handling and Troubleshooting in xfNetLink .NET

Troubleshooting Techniques

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
```

For example:

```
dbr -d xfpl 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 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 Traditional Synergy Programs](#)” chapter of *Synergy 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 10.1.1 ***  
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 Traditional Synergy Programs” chapter of *Synergy Tools*.)

Error Handling and Troubleshooting in xfNetLink .NET

Troubleshooting Techniques

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 Traditional Synergy Programs” chapter of *Synergy 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 (**dfs.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 <i>xfNetLink</i> clients should be compressed.	“Configuring Compression” on page 3-21
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 3-32
XFPL_LOGFILE	A filename for the log file.	XFPL_LOGFILE on page 3-32

Configuration Settings

xfServerPlus

xfpl.ini File Settings (continued)		
Use this setting	To specify this	For more information
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 3-33
XFPL_SESS_INFO	The level of session logging.	XFPL_SESS_INFO on page 3-33
XFPL_FUNC_INFO	The level of function logging.	XFPL_FUNC_INFO on page 3-35
XFPL_DEBUG	Debugging information should be written to the log.	XFPL_DEBUG on page 3-38

- 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 3-18
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 2-44

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 <i>xfServerPlus</i> is running.	"Specifying the Host Name and Port Number" on page 4-4
XF_REMOTE_PORT	Port that <i>xfServerPlus</i> is running on.	
XF_RMT_TIMEOUT	Call time-out for regular session or debug.	"Call time-out" on page 4-6
XF_RMT_DBG_TIMEOUT	Session time-out for running in debug mode.	"Connect session time-out" on page 4-6
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 4-7

xfNetLink Java

xfNetLink Java uses settings in the xfNetLink Java properties file (named **xfNetLnk.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 8-3](#) and [“Setting Up a Pooling Properties File” on page 8-24](#).

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 8-6
xf_RemotePort	Port that xfServerPlus is running on.	
xf_DebugOutput	Information useful in debugging should be logged.	“Specifying Logging Options” on page 8-7
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 8-8
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 8-8
xf_DebugSessionConnectTimeout	Amount of time xfNetLink should wait for acknowledgment from the logic server in xfServerPlus when running in debug mode.	

xfNetLink Java Properties File Settings (continued)		
Use this setting	To specify this	For more information
xf_SessionLingerTimeout	Amount of time xfNetLink should wait for a return from a remote call.	"Call time-out" on page 8-9 "Specifying Encryption Options" on page 8-9
xf_SSLEncertFile	The path and filename of the keystore file to use for encryption.	
xf_SSLPassword	Password associated with the keystore file.	

Configuration Settings

xfNetLink Java

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 8-27
maxPool	The maximum number of connections to be maintained in the pool.	
propertiesFile	The path and filename of the <i>xfNetLink</i> Java properties file to use. Required.	“Specifying the xfNetLink Java properties file to use” on page 8-27
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 8-27
poolLogFile	The filename that logging information should be written to.	“Specifying logging options” on page 8-28
poolLogLevel	The level of logging desired. Possible values are none, error, and all. If <i>poolLogFile</i> is specified and <i>poolLogLevel</i> is not specified, error level logging will take place.	
connectWaitTimeout	Number of seconds the <i>getConnection()</i> method will wait for a connection from the pool.	“Specifying time-out values” on page 8-29
initializationTimeout	Number of seconds <i>xfNetLink</i> should 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 method specified with the <i>initializationMethodID</i> setting.	

Pooling Properties File Settings (continued)		
Use this setting	To specify this	For more information
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 8-29
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 .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 11-3](#) 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 11-4
port	Port that xfServerPlus is running on.	
logging	The type of information you want logged.	“Creating and Editing Configuration Files” on page 11-4 and “Using Client-Side Logging” on page 12-7
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 11-4
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.	

Data Type Mapping

This appendix includes data type mapping information for *xfNetLink* Java and *xfNetLink* .NET.

xfNetLink Java

Parameter and return value type mapping

The tables in this section list the supported data types and show how parameter and return value data types are mapped from Synergy to Java when you generate Java class wrappers.

Note that for primitive Java data types, return values and “in” parameters are mapped as listed in the tables, but “out” and “in/out” parameters 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” parameter is mapped to a `DoubleHolder`. (See the `Java org.omg.CORBA` class for more information on holder classes.)



Type mapping varies depending on the version specified with the “Generate classes as version” option in Workbench (or the **genjava -c** option). See “[The genjava Utility](#)” on [page 7-14](#) for more information on this option.

Mapping when classes are generated for version 1.5 compatibility

For decimal, implied-decimal, and integer data types, you can choose to coerce the data to a non-default type on the Java side by selecting the desired data type in the Coerced type field when defining methods in the MDU. (See [page 2-27](#).) If you are using attributes instead of the MDU, see the description of the `cType` property on [page 2-11](#) for instructions on specifying non-default type mapping. Type coercion is supported only when the “Generate classes as version” option in Workbench is set to 1.5 (or **genjava** is run with the **-c 1.5** option). 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 Java data type” column will be used when “Default” is specified for the coerced type in the MDU.

Data Type Mapping

xfNetLink Java

xfNetLink Java Parameter and Return Value Type Mapping (v1.5 Compatibility)			
Data type in SMC	Size in SMC	Default Java data type	Non-default Java types available
^VAL ^a	N/A	int	N/A
Alpha	≤ 65,535	String ("in" parameters and return values) StringBuffer ("out" and "in/out" parameters)	N/A
Binary (handle) ^b	N/A	generic ArrayList	N/A
Decimal (d)	1, 2	byte	byte, short, int, long, Boolean, DateTime (Calendar), decimal (BigDecimal)
	3, 4	short	
	5–9	int	
	10–18	long	
	19–28	BigDecimal ("in" parameters and return values) FixedHolder ("out" and "in/out" parameters)	
Enumeration	N/A	enum type	N/A
Handle ^b	N/A	String ("in" parameters and return values) StringBuffer ("out" and "in/out" parameters)	N/A
Implied-decimal (d.)	≤ 16.*	double	decimal (BigDecimal), double, float
	≥ 17.*	BigDecimal ("in" parameters and return values) FixedHolder ("out" and "in/out" parameters)	
Integer	1	byte	byte, short, int, long, Boolean
	2	short	
	4	int	
	8	long	
System.String	N/A	String ("in" parameters and return values) StringBuffer ("out" and "in/out" parameters)	N/A

a. Return value only.

b. Parameter only.

Mapping when classes are generated for version 1.2 compatibility

xfNetLink Java Parameter and Return Value Type Mapping (v1.2 Compatibility)		
Data type in SMC	Size in SMC	Java data type
^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
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

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 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 in Java when the “Generate classes as version” option in Workbench is set to 1.5 (or **genjava** is run with the **-c 1.5** option). 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 Type Mapping

xfNetLink Java

data, take care to select a non-default type that is suitable for the size of the data that will be placed in the property. The data type in the “Default Java data type” column is used when “Default” is specified for the coerced type in Repository.

Some field type mappings vary depending on the version specified with the “Generate classes as version” option in Workbench (or the **genjava -c** option). These variations are noted in the table.

xfNetLink Java Field Type Mapping			
Field data type in Repository	Format	Default Java data type	Non-default Java data types available
Alpha	N/A	String (“in” parameters) StringBuffer (“out” and “in/out” parameters)	N/A
AutoSeq	N/A	long	N/A
AutoTime	N/A	Calendar (if -c 1.5) Date (if -c 1.2)	N/A
Binary	N/A	byte array (if -c 1.5) String (if -c 1.2)	N/A
Boolean	N/A	Boolean (if -c 1.5) byte (if -c 1.2)	N/A
Date ^a	YYMMDD YYYYMMDD YYJJJ YYYYJJJ	Calendar (if -c 1.5) Date (if -c 1.2)	N/A
	YYPP YYYYPP	String	
Decimal (no precision)	N/A	Mapping is the same as for decimal parameters; see the table on page B-2 (if -c 1.5) or page B-3 (if -c 1.2).	byte, short, int, long, Boolean, decimal (BigDecimal)
Decimal (with precision)	N/A	Mapping is the same as for implied-decimal parameters; see the table on page B-2 (if -c 1.5) or page B-3 (if -c 1.2).	double, decimal (BigDecimal)
Enum	N/A	Enum type for -c 1.5 Not supported for -c 1.2	N/A

xfNetLink Java Field Type Mapping			
Field data type in Repository	Format	Default Java data type	Non-default Java data types available
Integer	N/A	Mapping is the same as for parameters; see the table on page B-2 (if -c 1.5) or page B-3 (if -c 1.2).	byte, short, int, long, Boolean
Struct	N/A	N/A. Becomes a class (treated the same as a group).	N/A
Time	HHMM HHMMSS	Calendar (if -c 1.5) Date (if -c 1.2)	N/A
User	Date with ^CLASS^=YYYYMMDDHHMISS or YYYYMMDDHHMISSUUUUUU in the User data field	Calendar ^a	N/A
	Date with any other value in the User data field	String	
	Alpha		
	Binary		
	Numeric		

- a. When a Synergy routine sends a zero date to xfNetLink Java, a Date or Calendar 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.

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 2-11](#) 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, decimal, nullable decimal
Enumeration	enumeration	N/A
Handle ^b	string	N/A
Implied-decimal (d.)	decimal	double, float
Integer	length < 8 = int length = 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. The data type in the “Default C# data type” column will be used when “Default” is specified for the coerced type in the 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
AutoSeq	N/A	long	N/A
AutoTime	N/A	DateTime	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 10-19 .
Boolean	N/A	Boolean	N/A
Date	YYMMDD YYYYMMDD YYJJJ 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

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
Decimal (with precision)	N/A	decimal	double, float, nullable decimal
Enum	N/A	enumeration	N/A
Integer	N/A	length < 8 = int length = 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 or YYYYMMDDHHMISSUUUUUU 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
.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 `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. 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 3-2](#) for details.
4. Put **syncIt.dbl** on the client machine and edit the NAME identifier.
5. Compile and link **syncIt.dbl**.
6. Run **syncIt.dbr**. You’ll be prompted to enter the machine name and port where *xfServerPlus* is running.

xfNetLink 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 *xfNetLink* Java component generation tools. It sets the host and port, calls the `connect()` method to make a connection to *xfServerPlus*, and then calls the `hello_routine()` 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 *xfServerPlus*. It then calls the `hello_routine()` 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 *xfNetLink* 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("");
    int rtnval = 0;
```

xfNetLink Java Sample Code

Client Application (hello.jsp and hello.java)

```
String errmsg = "";
name = request.getParameter("UserName");

// Enter xfpl name or IP address and port number
helloexm.setxfHost("hostIP");
helloexm.setxfPort(2356);

// Connect to xfServerPlus and call hello method
helloexm.connect();
helloexm.hello_routine(name, message);

out.println("<br>");
out.println("The hello example returned message = ");
out.println(message);
out.println("<br>");

// shut down xfpl connection
helloexm.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_routine(name, message);
            System.out.println("message = " + message);
            // Close down the xfpl 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 *xf*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 7-5](#) or [“Creating a Java JAR File from the Command Line” on page 7-11](#) for instructions.
4. If *xfServerPlus* is not already running, start it on the server machine. See [“Running xfServerPlus” on page 3-2](#) 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 8-1](#) if you need help with deployment.

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 DBL that enables you to automate the population and maintenance of the Synergy Method Catalog. There are two attributes, `xfMethod` and `xfParameter`, each of which has a number of properties, which are used to specify metadata about your Synergy routines.

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.

coerced type. A non-default data type that the Synergy data type is converted to on the client side.

COM+. An enhanced version of Microsoft's Component Object Model (an interoperability standard that allows component objects developed in different languages to call one another). Among other improvements, COM+ handles many resource management tasks that developers previously had to program. COM+ enables you to implement pooling for `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 .NET`, you must create a COM+ application and then add components to it.

component. See [Synergy component](#).

dbl2xml. 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.

interface. Methods are grouped into interfaces for inclusion in a Synergy component. The interface name becomes the class name in the generated component.

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 webpage that combines HTML and code written in Java. When a browser requests a JavaServer Page, the web server executes the embedded code, allowing the webpage to interact with databases and other programs. For more information about JSP, see <http://www.oracle.com/technetwork/java/javase/jsp/index.html> in the Java section of the Oracle website.

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. *x/ServerPlus* uses this value to look up the routine to call. On *x/NetLink Synergy*, 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 *x/NetLink Java* and *x/NetLink .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. Network connection ID. See below.

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](#).

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.

Synergy assembly. Refers to a .NET assembly that references Synergy methods.

Glossary

Synergy component. A general term that can refer to a Synergy JAR file or assembly.

Synergy method. A Synergy function or subroutine that has been made available for use with *xfNetLink*. Synergy methods must be contained in an ELB or shared image and must be defined in the Synergy Method Catalog. Not to be confused with Synergy object-oriented methods.

SynergyWebProxy. An *xfNetLink* Java class that is instantiated in order to establish a connection between *xfNetLink* and *xfServerPlus* and to make calls to Synergy routines via *xfServerPlus*.

type coercion. See [coerced type](#).

utility methods. Refers to the standard methods that are included in every Synergy JAR file and assembly. These methods enable you to perform utility functions such as connecting, setting a time-out value, and running a debug session.

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 (.NET) 11-33
- activation method (Java) 8-33
- AddRow() method (.NET) 11-39
- alternate interface names 1-8
 - xfNetLink Java 7-8, 7-12
 - xfNetLink .NET 10-10, 10-15
- API
 - documentation, creating for client. *See* documentation, creating
 - xfNetLink Synergy 6-1 to 6-28
 - xfServerPlus 1-30 to 1-36
- application configuration file 11-3 to 11-8
 - creating 11-4 to 11-8
 - general information 11-3 to 11-4
 - naming 11-4
 - overriding settings at runtime 11-4
 - program defaults 11-3
 - using with Visual Studio project 11-5
- application, deploying. *See* client application, deploying; server application, deploying
- arguments. *See* data; parameters
- ArrayList
 - passing with xfNetLink Java 1-16
 - passing with xfNetLink .NET 1-16
 - System.Collections.ArrayList class 1-16
 - using to pass binary data (Java) 8-17 to 8-19
 - using to return structure collection
 - parameter 1-15 to 1-16
- arrays
 - defining in SMC 2-33
 - larger than 64K 1-18, 5-3
 - multi-dimensional 2-33
 - 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) 8-15
 - See also* ArrayList; parameters
- assemblies
 - configuration file 11-3 to 11-8
 - creating from command line 10-14 to 10-22
 - creating in Workbench 10-6 to 10-12

- deploying on client 11-1 to 11-2
- excluding structure fields from 1-10
- signing 10-8, 10-18, 10-25 to 10-26
- using in an application 11-9 to 11-24
- See also* C# classes; xfNetLink .NET

- AssemblyInfo.cs file
 - creation of 10-11, 10-17
 - editing 10-26 to 10-27
- 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 2-13
 - xfParameter 2-13 to 2-16

B

- base channel number 1-7
- batch file
 - xfNetLink Java 7-16
 - xfNetLink .NET 10-21
- BigDecimal (Java) 2-27, 2-32
 - See also* data type mapping
- binary data, passing
 - general information 1-18 to 1-19
 - structure fields (.NET) 10-19
 - xfNetLink Java 8-17 to 8-19
 - xfNetLink .NET 11-22 to 11-23
- byte array, passing
 - xfNetLink Java 8-17 to 8-19
 - xfNetLink .NET 11-22 to 11-23

C

- C# classes 10-6 to 10-25
 - alternate names for 10-10, 10-15
 - API documentation for 10-7, 10-18, 10-27 to 10-29
 - binary fields, converting 10-19
 - creating project for in Workbench 10-6 to 10-10
 - editing 10-26 to 10-27
 - generating from command line 10-14 to 10-21
 - generating from Workbench 10-11 to 10-12
 - generating properties vs. fields 10-9, 10-19
 - multiple copies of 11-12 to 11-13

Index

D

- namespace, specifying 10-8, 10-18
- overview of generated classes 10-23 to 10-25
- See also* assemblies; xfNetLink .NET
- C# interfaces 10-23, 11-12, 11-35
- cacerts file 3-25
- Calendar class (Java) 2-27, 2-32
- See also* data type mapping
- call time-out
 - xfNetLink Java 8-9, 8-19, 8-26
 - xfNetLink .NET 11-6, 11-23 to 11-24
 - xfNetLink Synergy 6-15
 - xfServerPlus 1-31
- calling Synergy routines
 - from Java client 8-10 to 8-14
 - from .NET client 11-9 to 11-24
 - from Synergy client 5-1 to 5-3
- CanBePooled() method (.NET) 11-33
- “Cannot load random state” error 3-23
- catalog. *See* Synergy Method Catalog
- certificate file for encryption 3-24
- chaining 1-3
- Changed property (.NET) 11-16
- “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
- classes. *See* C# classes; Java class wrappers
- classpath, setting 7-5
- cleanup method
 - Java pooling 8-34
 - registering with xfServerPlus 1-34 to 1-36
- Cleanup() method
 - registering with xfServerPlus 1-34 to 1-36
 - xfNetLink .NET 11-34
- client application, deploying
 - xfNetLink Java 8-1 to 8-2
 - xfNetLink .NET 11-1 to 11-2
 - xfNetLink Synergy 4-2
- client-side logging. *See* logging
- Clone() method (.NET) 11-15, 11-39
- closing connection to xfServerPlus
 - xfNetLink Java 8-14
 - xfNetLink .NET 11-12
 - xfNetLink Synergy 6-17
- coercing data types. *See* type coercion
- collection. *See* structure collection parameter
- collectionType property 2-15
- comments
 - documentation 2-20
 - in C# source files. *See* C# classes: API documentation for
 - in Java source files. *See* Javadoc
 - in pooling properties file 8-25
 - in Synergy source files 2-20
 - in synrc file 2-46, 3-19
 - in xfNetLink properties file 8-4
- comparing structure data (.NET) 11-15 to 11-17
- Component Information dialog box (Workbench)
 - Java project 7-6 to 7-9
 - .NET project 10-6 to 10-10
- components. *See* assemblies; JAR files
- compression of data 3-21
- configuration file. *See* application configuration file
- connect time-out
 - xfNetLink Java 8-8
 - xfNetLink .NET 11-6
 - xfNetLink Synergy 4-6, 6-19
- connect() method
 - xfNetLink Java 8-12, 8-35
 - xfNetLink .NET 11-10, 11-37
- connection pooling. *See* Java pooling
- connection to xfServerPlus, closing
 - xfNetLink Java 8-14
 - xfNetLink .NET 11-12
 - xfNetLink Synergy 6-17
- converting data. *See* data type mapping
- cType property
 - method 2-11
 - parameter 2-15

D

- D_NO_GLOBAL_DATA 1-5
- data
 - accessing remotely from xfServerPlus 3-49 to 3-52
 - global 1-3, 1-5
 - larger than 64K 1-13 to 1-15
 - packed 6-24
 - supported types 2-32
 - variable length 1-13 to 1-17, 5-3
 - See also* data type mapping; parameters
- data compression 3-21
- data type conversion. *See* data type mapping

- data type mapping
 - xfNetLink Java B-1 to B-5
 - xfNetLink .NET B-6 to B-8
 - See also* type coercion
- dataTable property 2-16
- DataTables (.NET) 10-24, 11-17 to 11-21
 - column caption 11-19
 - column names 11-18
 - defining in MDU 2-33
 - defining with attribute 2-16
 - examples 11-20
 - methods 11-19, 11-39
 - non-nullable fields in 11-18
 - overview 10-24
 - primary key 11-19
 - read-only fields in 11-18
- DateTime 2-27, 2-32
 - xfNetLink .NET B-7, B-8
 - See also* Calendar class (Java)
- dbl2xml utility 2-3, 2-5
 - See also* Synergy Tools
- dbs.exe 1-5, 3-30, 3-43
 - See also* Synergy Tools
- Deactivate() method (.NET) 11-33
- deactivation method (Java) 8-33
- debug mode, running xfServerPlus in 3-43
 - from Java client 9-10 to 9-14
 - from .NET client 12-10 to 12-14
 - from Synergy client 5-4 to 5-8
- debugging 3-43 to 3-47
 - debugInit() method
 - xfNetLink Java 8-35, 9-11
 - xfNetLink .NET 11-37, 12-11
 - debugStart() method
 - xfNetLink Java 8-35, 9-12
 - xfNetLink NET 11-37, 12-12
 - overview 3-43
 - %RX_DEBUG_INIT function 6-4
 - %RX_DEBUG_START function 6-5 to 6-6
 - setting XFPL_DEBUG 3-38 to 3-40
 - test skeletons, generating 1-21 to 1-29
 - with Telnet 3-44 to 3-47
 - xfNetLink Java 9-7 to 9-14
 - xfNetLink .NET 12-7 to 12-14
 - xfNetLink Synergy 5-4 to 5-9
 - See also* errors; logging; testing; xfServerPlus log

- debugInit() method
 - xfNetLink Java 8-35, 9-11
 - xfNetLink .NET 11-37, 12-11
- debugStart() method
 - xfNetLink Java 8-35, 9-12
 - xfNetLink NET 11-37, 12-12
- default port, xfServerPlus 3-2
- defaultsmc.xml file 2-38
- delayed signing of assemblies 10-8, 10-18, 10-25
- deploying applications. *See* client application, deploying; server application, deploying
- disconnect() method
 - xfNetLink Java 8-14, 8-35
 - xfNetLink .NET 11-12, 11-37
- disconnecting from xfServerPlus
 - xfNetLink Java 8-14
 - xfNetLink .NET 11-12
 - xfNetLink Synergy 6-17
- dllhost.exe.config file 11-4
- documentation comments 2-20
- documentation, creating
 - adding comments to Synergy source files 2-20
 - for .NET assemblies 10-7, 10-18, 10-27 to 10-29
 - Javadoc 7-7, 7-15, 7-20 to 7-22

E

- elb property 2-9
- ELBs 1-3 to 1-5
 - calling other ELBs 1-3
 - closing 1-31
 - defining in MDU 2-25
 - defining logicals for 1-4, 2-53
 - 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 2-13
- encryption 3-22 to 3-30
 - and xfNetLink Java logging 9-7
 - and xfNetLink .NET logging 12-7
 - and xfNetLink Synergy logging 5-8
 - and xfServerPlus logging 3-22
 - certificate file 3-24
 - disabling 3-23
 - genCert utility 3-26
 - Java client set-up 3-25 to 3-27

Index

F

- master vs. slave 3-22
 - .NET client set-up 3-28
 - OpenSSL installation 3-23
 - specifying data for slave encryption 3-29
 - Synergy client set-up 3-25
 - xfNetLink Java properties file settings 8-9
 - xfServerPlus machine set-up 3-23
 - enumerations
 - defining in MDU 2-26, 2-30
 - passing as parameters and return values 1-12
 - xfNetLink Java 8-16
 - xfNetLink .NET 11-21
 - environment variables
 - RPSDAT. *See* repository files, specifying location
 - RPSMFIL. *See* repository files, specifying location
 - RPSTFIL. *See* repository files, specifying location
 - SYNCSOFT 10-12, 10-21
 - SYNSSL_RAND 3-23
 - SYNSSLLIB 3-24
 - XF_REMOTE_HOST 4-4
 - XF_REMOTE_PORT 4-4
 - XF_RMT_DBG_TIMEOUT 4-6
 - XF_RMT_TIMEOUT 4-6
 - XF_RMTCONN_TIMEOUT 4-6
 - XFBOOTCLASSPATH 7-10, 7-16
 - XFEXTDIRS 7-10, 7-16
 - XFNLS_LOGFILE 4-7
 - XFPL_DBR. *See* *Environment Variables & System Options*
 - XFPL_INIPATH 3-18 to 3-20
 - XFPL_SMCPATH 2-44 to 2-47
 - Equals() method (.NET) 11-15, 11-39
 - \$ERR_CHNUSE error 1-7
 - \$ERR_FILOPT error 1-6
 - \$ERR_XFNOCALL error 6-3
 - \$ERR_XFNOINIT error 6-5
 - error handling. *See* errors
 - error log. *See* logging; xfServerPlus log
 - errors
 - “Channel is in use” 1-7, 1-33
 - \$ERR_CHNUSE 1-7
 - \$ERR_FILOPT 1-6
 - \$ERR_XFNOCALL 6-3
 - \$ERR_XFNOINIT 6-5
 - handling (general information) 1-19
 - “Invalid operation for file type” 1-6
 - service specific error 14 3-24
 - signaled by %RX_START_REMOTE 6-20 to 6-22
 - signaled by %RXSUBR 6-26 to 6-27
 - xfNetLink Java 9-1 to 9-6
 - xfNetLink .NET 12-1 to 12-6
 - xfNetLink Synergy 5-3 to 5-4, 6-20 to 6-22, 6-26 to 6-27
 - xfServerPlus 3-15, 3-41 to 3-42
 - See also* debugging; exceptions
 - exceptions
 - handling for xfNetLink Java 9-1 to 9-6
 - handling for xfNetLink .NET 12-1 to 12-6
 - xfJCWException 8-45
 - xfPoolException 8-46
 - See also* errors
 - “Excluded by web” flag 1-10
 - exporting methods from SMC 2-38 to 2-40
- ### F
- fatal errors. *See* errors; exceptions
 - fields vs. properties, generating for .NET structure
 - members 10-9, 10-19
 - firewall, when debugging via Telnet 3-44
 - format property
 - method 2-12
 - parameter 2-15
 - functions. *See* Synergy routines
- ### G
- GAC 11-1, 11-2
 - genCert utility 3-26
 - gens utility 10-17 to 10-21
 - genjava utility 7-14 to 7-16
 - gensyn utility 1-25 to 1-26
 - genxml utility
 - test skeletons 1-24
 - xfNetLink Java 7-11 to 7-14
 - xfNetLink .NET 10-14 to 10-17
 - getConnection() method
 - xfNetLink Java 8-12, 8-35
 - xfNetLink .NET 11-10, 11-38
 - getEnumeration() method (Java) 8-36
 - getInstance() method (Java) 8-42
 - getIntHolderValue() method (Java) 8-36
 - getIntValue() method (Java) 8-36
 - getPoolName() method (Java) 8-36
 - GetRow() method (.NET) 11-39
 - GetRows() method (.NET) 11-40
 - getSSLCertFile() method (Java) 8-37

- getSSLPassword() method (Java) 8-37
- getSynergyWebProxy() method (Java) 8-37
- getUserString() method
 - xfNetLink Java 8-20, 8-37
 - xfNetLink .NET 11-24, 11-38
- getxfHost() method (Java) 8-37
- getxfLogfile() method (Java) 8-37
- getxfPort() method (Java) 8-37
- global data 1-3
 - excluding from tools.def file 1-5
- group arguments 2-7, 2-17

H

- handle. *See* memory handle
- help, obtaining xii
- host name, defining on client
 - xfNetLink Java 8-6, 8-12, 8-37
 - xfNetLink .NET 11-6
 - xfNetLink Synergy 4-4, 6-18
- host port number, defining on client
 - xfNetLink Java 8-6, 8-12, 8-37
 - xfNetLink .NET 11-7
 - xfNetLink Synergy 4-4, 6-18

I

- id property 2-10
- importing methods to SMC 2-38 to 2-40
- .INCLUDE directive 2-16, 2-17
- initialization method (Java) 8-33
- initialize time-out
 - xfNetLink Java 8-26, 8-29
 - xfNetLink .NET 11-6
- Initialize() method (.NET) 11-32
- INotifyPropertyChanged class 10-9, 10-19
- interface property 2-8
- interfaces, naming 2-24
 - See also* alternate interface names
- “Invalid operation for file type error” 1-6
- IP address. *See* host name, defining on client

J

- JAR files
 - creating from command line 7-11 to 7-17
 - creating in Workbench 7-5 to 7-10
 - deploying on client 8-1
 - excluding structure fields from 1-10
 - using classes in 8-10 to 8-14

- version compatibility 7-7, 7-15
- xfnljav.jar 7-5
 - See also* Java class wrappers
- Java class wrappers 7-5 to 7-20
 - alternate names for 7-8, 7-12
 - creating documentation for 7-7, 7-15, 7-20 to 7-22
 - creating project for in Workbench 7-6 to 7-9
 - editing the generated files 7-20
 - generating from command line 7-11 to 7-16
 - generating from Workbench 7-9
 - package name, specifying 7-7, 7-14
 - understanding the generated files 7-18 to 7-20
 - version compatibility 7-7, 7-15
 - See also* JAR files; xfNetLink Java
- Java classes 8-40 to 8-46
 - See also specific Java classes*
- Java component project, creating 7-6 to 7-9
- Java JAR file. *See* JAR files
- Java pooling 8-21 to 8-34
 - debugging remote routines and 3-43, 3-45
 - error logging 8-22, 8-28
 - overview 8-21 to 8-22
 - pool ID 8-25
 - pool maintenance 8-31 to 8-32
 - pool size 8-27
 - properties file 8-24 to 8-30
 - returning pool to minimum size 8-31
 - reusing connections 8-21, 8-27, 8-34
 - shutting down the pool 8-32, 8-43
 - support methods 8-29, 8-32 to 8-34
 - SWPManager class 8-42
 - time-outs 8-29
 - updating properties files 8-31
 - usePool() method 8-23, 8-39
 - using multiple pools 8-25
 - writing code that uses pooled
 - connections 8-22 to 8-24
 - xfPoolException class 8-46
 - See also* pooling properties file
- Javadoc
 - deploying 8-2
 - generating 7-7, 7-15, 7-20 to 7-22
- JavaServer Pages 7-1
 - code sample D-1 to D-5
- JDK, supported versions 7-1

Index

K

K

KEEPALIVE timer value 1-31
key file. *See* strong name key file

L

length property
 method 2-10
 parameter 2-14
license requirements for xfServerPlus xii
“log on as a batch job” user right, error if not set 6-21
“log on locally” user right 3-2
logging
 xfNetLink Java 8-7, 8-28, 9-7 to 9-9
 xfNetLink .NET 11-7, 12-7
 xfNetLink Synergy 4-7, 5-8 to 5-9
 xfServerPlus 3-30
 See also xfServerPlus log
logicals, defining for ELBs 1-4, 2-53
 See also environment variables

M

master encryption 3-22
MDU. *See* Method Definition Utility
mdu.dbr 2-48 to 2-52
 See also Method Definition Utility
memory handle
 using to pass binary data
 general information 1-18 to 1-19
 xfNetLink Java 8-17 to 8-19
 xfNetLink .NET 11-22 to 11-23
 using to pass large parameters 1-13 to 1-14
 using to pass variable-length parameters 1-13 to 1-14
 using to return structure collection 1-15 to 1-16
Method Definition Utility 2-22 to 2-37
 changing method ID 2-28
 command line syntax 2-48 to 2-52
 creating new methods 2-22 to 2-34
 date last updated field 2-35
 defining enumeration parameters 2-30
 defining parameters 2-28 to 2-34
 defining structure parameters 2-30
 deleting data 2-36
 exporting methods 2-38 to 2-40, 2-49
 importing methods 2-38 to 2-40, 2-49, 2-50
 modifying entries 2-35
 remote SMC, using with 2-45, 2-48
 resequencing parameters 2-34

 searching for data in 2-37
 specifying alternate location for SMC files 2-44
 specifying method ID 2-28
 verifying structure sizes 2-41
 See also Synergy Method Catalog
method IDs
 defining in MDU 2-28
 defining with attribute 2-10
 passing to %RXSUBR 6-23
 vs. method name when making remote calls 2-2
 See also Method Definition Utility
methods
 included in Java JAR file 8-35 to 8-39
 included in .NET assembly 11-37 to 11-40
 pooling support
 xfNetLink Java 8-32 to 8-34
 xfNetLink .NET 11-31 to 11-34
 Synergy. *See* Method Definition Utility; Synergy
 Method Catalog

modular code 1-1
multiple copies of C# classes 11-12 to 11-13
multiple SMCs 2-42 to 2-47
multiple xfpl.ini files 3-17

N

name property
 method 2-9
 parameter 2-13
namespace, specifying 10-8, 10-18
.NET component project, creating 10-6 to 10-10
.NET Framework
 installing redistributable 11-2
 regsvcs utility 11-28
 sdkvars.bat file 10-11, 10-21
 setting up environment 10-11, 10-21
 sn.exe 10-26
 supported version 10-1
 System.Data.DataTable class 11-17
 vcvarsall.bat file 10-22
 vsvars32.bat file 10-21
.NET pooling 11-25 to 11-36
 ASP and 11-35
 building assembly for pooling 10-22
 changing pool configuration 11-31
 checking the pool status 11-31
 creating poolable objects 11-26
 debugging remote routines and 3-43, 3-45

- initialize time-out setting 11-6
- overview 11-25 to 11-26
- pool return setting 11-7
- pool size 11-29
- reusing objects 11-25
- setting up 11-27 to 11-31
- starting the pool 11-31
- support methods 11-31 to 11-34
- writing code that uses pooled objects 11-34
- network connection ID 5-2, 6-18
 - invalid 6-27
- nullable DateTime (.NET) B-7
 - defining in MDU 2-27, 2-32
 - defining with attribute 2-12, 2-15
- nullable property
 - method 2-12
 - parameter 2-15

O

- object pooling. *See* .NET pooling
- OpenSSL. *See* encryption
- OpenVMS
 - servstat program 3-12, 3-13
 - unsupported routines 1-4
- Original property (.NET) 11-16
- overlays in repository structures 1-9 to 1-10

P

- package name, specifying 7-7, 7-14
- packed data 6-24
- 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 1-18
 - binary 1-18 to 1-19
 - coercing data type 2-32, B-6 to B-8
 - DataTable 10-24, 11-17 to 11-21
 - defining in MDU 2-28 to 2-34
 - generating out vs. ref in C# code 10-8, 10-19
 - group 2-7, 2-17
 - larger than 64K 1-13 to 1-15, 1-18
 - xfNetLink Synergy 5-3
 - mapping of data types B-1 to B-8

- maximum number allowed 2-28
- optional 2-34, 8-13, 11-11
- packed 6-24
- passing enumerations as 1-12
- passing structures as 1-8 to 1-11
- ^REF 2-34, 6-24
- required 2-34, 8-13, 11-11
- sequence of 2-34
- String class 1-14
- structure collection 1-15 to 1-16, 2-33
- supported by %RXSUBR 6-24
- supported data types 2-32
- System.Collections.ArrayList class 1-16, 2-33
- System.String class 1-14
- type mapping B-1 to B-8
- ^VAL 2-34, 6-24
- variable length 1-13 to 1-17, 5-3
- .pem file 3-4, 3-24
- poolable method (Java) 8-34
- pooling properties file 8-24 to 8-30
 - comments in 8-25
 - multiple pools 8-25
 - naming 8-24
 - placement of 8-24
 - pool ID 8-25
 - rereading while pool is active 8-31
 - sample 8-30
 - settings 8-25 to 8-30
 - activationMethodID 8-29
 - cleanupMethodID 8-29
 - connectWaitTimeout 8-29
 - deactivationMethodID 8-29
 - initializationMethodID 8-29
 - initializationTimeout 8-29
 - maxPool 8-27
 - minPool 8-27
 - poolableMethodID 8-29
 - poolLogFile 8-28
 - poolLogLevel 8-28
 - poolReturn 8-27
 - propertiesFile 8-27
- pooling. *See* Java pooling; .NET pooling
- port number range
 - xfNetLink Java 9-12
 - xfNetLink .NET 12-11

Index

R

- port number, xfServerPlus
 - default 3-2
 - setting
 - OpenVMS 3-11, 3-13
 - UNIX 3-9
 - Windows 3-3, 3-6
 - See also* host port number, defining on client
 - precision property
 - method 2-10
 - parameter 2-14
 - Professional Series Workbench. *See* Workbench
 - properties file. *See* pooling properties file; xfNetLink Java properties file
 - properties vs. fields, generating for .NET structure members 10-9, 10-19
 - PropertyChanged event 10-9, 10-19
 - protocol version 3-40
 - pseudo arrays 2-33
- ## R
- RCB. *See* routine call block
 - rcbid 6-16
 - rd.log file 3-46
 - read-only properties
 - xfNetLink Java 7-8, 7-15, 8-14
 - xfNetLink .NET 10-9, 10-19, 11-13
 - real arrays 2-33
 - ^REF parameters
 - defining in MDU 2-34
 - passing to %RXSUBR 6-24
 - registering
 - assembly in GAC 11-1, 11-2
 - cleanup method with
 - XFPL_REGCLEANUP 1-34 to 1-36
 - xfServerPlus on Windows 3-6
 - regsvcs utility 11-28
 - remote data access from xfServerPlus 3-49 to 3-52
 - repository enumerations. *See* enumerations
 - repository fields. *See* structures
 - repository files, specifying location
 - assembly 10-8, 10-15
 - JAR file 7-7, 7-12
 - test skeletons 1-23
 - repository structures. *See* structures
 - request for session time-out
 - xfNetLink Java 8-8
 - xfNetLink .NET 11-6
 - xfNetLink Synergy 4-5
 - required parameters for Java and .NET 2-7, 2-34
 - resetPoolProperties() method (Java) 8-31, 8-43
 - returnToMinimum() method (Java) 8-31, 8-43
 - routine call block 5-3, 6-16
 - routines. *See* Synergy 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*
 - rsynd.pem file 3-24
 - running xfServerPlus 3-2 to 3-13
 - OpenVMS 3-11 to 3-13
 - UNIX 3-8 to 3-9
 - Windows 3-2 to 3-7
 - %RX_CONTINUE function 6-2
 - %RX_DEBUG_INIT function 5-5, 6-4
 - %RX_DEBUG_START function 5-6, 6-5 to 6-6
 - RX_GET_ERRINFO subroutine 6-7 to 6-8
 - RX_GET_HALTINFO subroutine 6-9 to 6-10
 - %RX_RMT_ENDIAN function 6-11
 - %RX_RMT_INTSIZE function 6-12
 - %RX_RMT_OS function 6-13
 - %RX_RMT_SYSINFO function 6-14
 - %RX_RMT_TIMEOUT function 6-15
 - RX_SETRMETFNC subroutine 6-16
 - RX_SHUTDOWN_REMOTE subroutine 6-17
 - %RX_START_REMOTE function 6-18 to 6-22
 - status codes 6-20 to 6-22
 - rxapi.def file 5-1, 6-18
 - rxerr.def file 5-1, 6-7
 - %RXSUBR routine 6-23 to 6-27
 - runtime errors signaled by 6-26 to 6-27
 - using to make remote call 5-1 to 5-2

S

- Sandcastle application 10-29
- sdkvars.bat file 10-11, 10-21
- server application, deploying 3-47
- service runtime 1-5, 3-30, 3-43
 - See also Synergy Tools*
- servstat program 3-13
 - displaying xfServerPlus status 3-12
 - purging free pool 3-13
- “set” methods (Java)
 - compared with properties file 8-5 to 8-6
 - using 8-12
 - See also individual methods*
- SET_XFPL_TIMEOUT subroutine 1-31
- setCallTimeout() method (.NET) 11-38
- setruiser 3-8, 3-49, 3-51
- setSSLCertFile() method (Java) 8-37
- setSSLPassword() method (Java) 8-38
- setUserString() method
 - xfNetLink Java 8-20, 8-38
 - xfNetLink .NET 11-24, 11-38
- setxfCallTimeout() method (Java) 8-19, 8-38
- setxfHost() method (Java) 8-38
- setxfLogfile() method (Java) 8-38
- setxfLogging() method (Java) 8-39
- setxfPort() method (Java) 8-39
- SFWINIPATH environment variable 3-43
- shareConnect() method
 - xfNetLink Java 8-12, 8-39
 - xfNetLink .NET 11-10, 11-38
- shared images. *See* ELBs
- SHELL routine 1-4
- shutdown() method (Java) 8-32, 8-43
- slave encryption 3-22
- SMC. *See* Synergy Method Catalog
- smc_elb.exe 2-53 to 2-56
- SMC/ELB Comparison utility 2-53 to 2-56
- sn.exe 10-26
- SPAWN routine 1-4
- starting xfServerPlus
 - OpenVMS 3-11 to 3-13
 - UNIX 3-8 to 3-9
 - Windows 3-2 to 3-7
- status codes
 - %RX_START_REMOTE 6-20 to 6-22
 - xfServerPlus 3-15
- stopping xfServerPlus
 - OpenVMS 3-13
 - UNIX 3-10
 - Windows 3-7 to 3-8
- strong name key file 10-8, 10-18, 10-25 to 10-26
- struct data type field in repository 1-8
- structfields 2-7
 - examples 2-17, 2-19
- structure classes, using
 - xfNetLink Java 8-14 to 8-16
 - xfNetLink .NET 11-13 to 11-17
- structure collection parameter 1-15 to 1-16
 - defining in MDU 2-33
 - defining with attribute 2-15
- structure property 2-16
- structures
 - attributing 1-8, 2-7
 - binary fields (.NET) 10-19
 - comparing data in (.NET) 11-15 to 11-17
 - defining in MDU 2-30
 - excluding fields from 1-10
 - field type mapping
 - xfNetLink Java B-3
 - xfNetLink .NET B-7
 - overlays, handling 1-9 to 1-10
 - passing as parameters 1-8 to 1-11
 - properties vs. fields for .NET 10-9, 10-19
 - read-only fields
 - xfNetLink Java 8-14
 - xfNetLink .NET 11-13
 - returning collection of 1-15 to 1-16
 - supported types 1-8
 - verifying sizes in SMC 2-41
 - See also* structfields
- support, obtaining technical xii
- SWPConnect class (Java) 8-40
- SWPManager class (Java) 8-42
 - using methods in 8-23, 8-31 to 8-32
- SYNCSOFT environment variable 10-12, 10-21
- Synergy code. *See* Synergy routines; Synergy server code
- Synergy components. *See* assemblies; JAR files
- Synergy Configuration Program
 - setting XFPL_INIPATH 3-18 to 3-19
 - setting XFPL_SMCPATH 2-45 to 2-46
 - starting xfServerPlus 3-3 to 3-5
 - stopping xfServerPlus 3-7 to 3-8

Index

T

- Synergy DBL data types
 - defining in MDU 2-26, 2-31
 - mapping to C# types B-6 to B-8
 - mapping to Java types B-1 to B-5
 - supported 2-32
 - Synergy Method Catalog 2-1 to 2-56
 - accessing files on remote machine 2-45, 2-48
 - creating new files 2-42 to 2-43
 - default location 2-42
 - exporting methods 2-38 to 2-40
 - importing methods 2-38 to 2-40
 - overview 2-1
 - populating. *See* attributes; Method Definition Utility
 - specifying when creating components
 - assembly 10-9, 10-15
 - JAR file 7-8, 7-12
 - specifying when creating test skeletons 1-23
 - updating 2-38 to 2-40
 - using logicals in 1-4, 2-53
 - using multiple 2-42 to 2-47
 - verifying entries against ELB 2-53 to 2-56
 - verifying structure sizes 2-41
 - See also* attributes; Method Definition Utility
 - Synergy .NET Configuration utility. *See* xfNetLink .NET Configuration utility
 - Synergy routines
 - calling from client
 - xfNetLink Java 8-10 to 8-14
 - xfNetLink .NET 11-9 to 11-24
 - xfNetLink Synergy 5-1 to 5-3
 - debugging 3-43 to 3-47
 - defining for use with xfNetLink 2-1 to 2-36
 - searching for in MDU 2-37
 - UI Toolkit routines 1-5 to 1-6
 - unsupported 1-3, 1-4
 - See also* Synergy server code
 - Synergy server code
 - debugging 3-43 to 3-47
 - defining routines for use with xfNetLink 2-1 to 2-36
 - ELB restrictions 1-3
 - enclosing in ELBs 1-3
 - modularizing 1-1
 - preparing for remote access 1-1 to 1-29
 - testing 1-21 to 1-29
 - UI Toolkit routines and 1-5 to 1-6
 - user interface elements in 1-5
 - Synergy/DE routines. *See* Synergy routines
 - synergy.ini file
 - configuring for xfNetLink Synergy 4-4 to 4-7
 - read by dbs 3-43
 - synnetcfg.exe 11-4
 - synrc file
 - comments in 2-46
 - setting XFPL_INIPATH in 3-19
 - setting XFPL_SMCPATH in 2-46
 - synrc.com, setting XFPL_INIPATH in 3-20
 - SYNSSL_RAND environment variable 3-23
 - SYNSSLIB environment variable 3-24
 - System.Collections.ArrayList class 1-16
 - System.Data.DataTable class 11-17
 - System.String class 1-14
- ### T
- technical support, obtaining xii
 - Telnet, using to debug remotely 3-44 to 3-47
 - test skeletons, generating 1-21 to 1-29
 - testing
 - Synergy server code 1-21 to 1-29
 - xfNetLink Java 9-9 to 9-10
 - xfNetLink .NET 12-9 to 12-10
 - xfNetLink Synergy 4-8
 - xfServerPlus 3-14
 - time-outs
 - in debug mode 5-5, 9-12, 12-11
 - KEEPALIVE 1-31
 - recorded in xfServerPlus log 3-34
 - using %RX_CONTINUE function 6-2
 - xfNetLink Java 8-7 to 8-9, 8-19, 8-29
 - xfNetLink .NET 11-6, 11-23 to 11-24, 11-29
 - xfNetLink Synergy 4-5 to 4-7, 6-15
 - xfServerPlus 1-31
 - Toolkit routines. *See* UI Toolkit routines
 - tools.def file 1-5
 - translating data. *See* data type mapping
 - troubleshooting. *See* debugging; errors; testing
 - type coercion
 - defining in MDU 2-27, 2-32
 - defining with attribute 2-11, 2-15
 - xfNetLink Java B-1 to B-5
 - xfNetLink .NET B-6 to B-8
 - See also* data type mapping
 - type mapping. *See* data type mapping; type coercion
 - type property 2-13

U

U_START subroutine 1-5
 UI Toolkit routines 1-5 to 1-6
 usePool() method (Java) 8-39
 user interface elements in Synergy server code 1-5
 utility methods
 xfNetLink Java 8-35 to 8-39
 xfNetLink .NET 11-37 to 11-40

V

^VAL parameters
 defining in MDU 2-34
 passing to %RXSUBR 6-24
 vcvarsall.bat file 10-22
 verifying repository structure sizes in SMC 2-41
 Visual Studio
 referencing Synergy assembly in project 11-9
 referencing xfnlnet.dll 11-9
 supported versions 10-1
 using application configuration file 11-5
 vcvarsall.bat file 10-22
 vsvars32.bat file 10-11, 10-21
 VMS. *See* OpenVMS
 vsvars32.bat file 10-11, 10-21

W

web.config file 11-4
 Windows application event log
 server-side logging 3-17, 3-30, 3-31, 5-3
 xfNetlink .NET logging 12-7
 Workbench
 Java component project 7-6 to 7-10
 .NET component project 10-6 to 10-12
 setting up the .NET environment 10-11
 SMC/ELB comparison, running 2-56
 test skeletons, generating 1-22 to 1-24

X

xercesImpl.jar file 7-5
 xf_DebugOutput setting 8-7, 9-7
 xf_DebugSessionConnectTimeout setting 8-8
 xf_LogFile setting 8-7, 9-7
 XF_REMOTE_HOST environment variable 4-5
 XF_REMOTE_PORT environment variable 4-5
 xf_RemoteHostName setting 8-6
 xf_RemotePort setting 8-6
 XF_RMT_DBG_TIMEOUT environment variable 4-6

XF_RMT_TIMEOUT environment variable 4-6
 XF_RMTCONN_TIMEOUT environment variable 4-6
 xf_SessionConnectTimeout setting 8-8
 xf_SessionLingerTimeout setting 8-9
 xf_SessionRequestTimeout setting 8-8
 xf_SSLErtFile setting 8-9
 xf_SSLErtPassword setting 8-9
 XFBOOTCLASSPATH environment variable 7-10, 7-16
 XFEXTDIRS environment variable 7-10, 7-16
 xjCWEException class 8-45
 xfMethod attribute 2-8 to 2-13
 xfNetLink Java
 ArrayList
 using to pass binary data 8-17 to 8-19
 using to return structure collection 1-15 to 1-16
 classes 8-40 to 8-46
 classpath, setting 7-5
 code sample D-1 to D-5
 data type mapping B-1 to B-5
 deploying application 8-1 to 8-2
 encryption set-up 3-25 to 3-27
 errors 9-1 to 9-6
 exception handling 9-1 to 9-6
 genCert utility 3-26
 host name, specifying 8-6, 8-12
 host port, specifying 8-6, 8-12
 importing required packages 8-11, 8-22
 logging 9-7 to 9-9
 making remote calls 8-10 to 8-14
 method reference 8-35 to 8-39
 minimum version requirements 7-1
 overview of tasks 7-3 to 7-4
 pooling 8-21 to 8-34
 properties file. *See* pooling properties file; xfNetLink
 Java properties file
 running xfServerPlus session in debug
 mode 9-10 to 9-14
 setting classpath 7-5
 system requirements 7-1
 System.Collections.ArrayList, passing 1-16
 System.String, passing 1-14
 technical overview 7-1
 testing 9-9 to 9-10
 time-outs 8-7 to 8-9, 8-19, 8-29
 type coercion B-1 to B-5
 See also JAR files; Java class wrappers; Java pooling

Index

X

- xfNetLink Java properties file 8-3 to 8-10
 - comments in 8-4
 - compared with “set” methods 8-5 to 8-6
 - connection pooling and 8-27
 - naming 8-4, 8-27
 - placement of 8-4
 - settings
 - xf_DebugOutput 8-7
 - xf_DebugSessionConnectTimeout 8-8
 - xf_LogFile 8-7
 - xf_RemoteHostName 8-6
 - xf_RemotePort 8-6
 - xf_SessionConnectTimeout 8-8
 - xf_SessionLingerTimeout 8-9
 - xf_SessionRequestTimeout 8-8
 - xf_SSLEntFile 8-9
 - xf_SSLEntPassword 8-9
 - specifying for pooling 8-27
- xfNetLink .NET
 - ArrayList, using to return structure
 - collection 1-15 to 1-16
 - configuration file 11-3 to 11-8
 - configuration utility 11-4 to 11-8, 11-27
 - data type mapping B-6 to B-8
 - deploying application 11-1 to 11-2
 - encryption set-up 3-28
 - error handling 12-1 to 12-6
 - Framework version in Workbench 10-11
 - host name, specifying 11-6
 - host port, specifying 11-7
 - logging 12-7
 - making remote calls 11-9 to 11-24
 - method reference 11-37 to 11-40
 - minimum version requirements 10-1
 - overview of tasks 10-3 to 10-5
 - pooling 11-25 to 11-36
 - running xfServerPlus session in debug
 - mode 12-10 to 12-14
 - system requirements 10-1
 - System.Collections.ArrayList, passing 1-16
 - System.String, passing 1-14
 - technical overview 10-2
 - testing 12-9 to 12-10
 - time-outs 11-6, 11-23 to 11-24, 11-29
 - type coercion B-6 to B-8
 - See also* assemblies; C# classes; .NET pooling
- xfNetLink .NET Configuration utility 11-4 to 11-8, 11-27
- xfNetLink Synergy
 - API 6-1 to 6-28
 - code sample C-1 to C-5
 - configuring 4-4 to 4-7
 - encryption set-up 3-25
 - error handling 5-3 to 5-4
 - logging 5-8 to 5-9
 - making remote calls 5-1 to 5-3
 - overview of tasks 4-2 to 4-3
 - running xfServerPlus session in debug
 - mode 5-4 to 5-8
 - technical overview 4-1 to 4-2
 - testing 4-8
 - time-outs 4-5 to 4-7, 6-15
- xfNetLnk.ini file. *See* xfNetLink Java properties file
- xfnljav.jar file 7-5
- xfnLJTest.class 9-9
- xfnlnet.dll 11-1, 11-2
 - referencing in Visual Studio project 11-9
- XFNLS_LOGFILE environment variable 4-7
- xfnlstst program 4-8
- xfParameter attribute 2-13 to 2-16
- xfpl_api library 1-30
- XFPL_BASECHAN setting 1-7
- XFPL_COMPRESS setting 3-21
- XFPL_DBR environment variable. *See Environment Variables & System Options*
- XFPL_DEBUG setting 3-38 to 3-40
- XFPL_FUNC_INFO setting 3-35 to 3-38
- XFPL_INIPATH environment variable 3-18 to 3-20
- XFPL_LOG setting 3-32
- XFPL_LOG subroutine 1-33
- XFPL_LOGFILE setting 3-32
- XFPL_LOGICAL setting 1-4
- XFPL_REGCLEANUP subroutine 1-34 to 1-36
- XFPL_SESS_INFO setting 3-33 to 3-35
- XFPL_SINGLELOGFILE setting 3-33
- XFPL_SMCPATH environment variable
 - setting for MDU 2-44
 - setting for xfServerPlus 2-44 to 2-47
- xfpl_tst.elb library 3-14, 4-8, 9-9
- xfpl.dbr 5-6, 9-12, 12-12

- xfpl.ini file 3-17 to 3-20
 - default location 3-17
 - defining logicals in 1-4
 - errors in 3-17
 - settings
 - XFPL_BASECHAN 1-7
 - XFPL_COMPRESS 3-21
 - XFPL_DEBUG 3-38 to 3-40
 - XFPL_FUNC_INFO 3-35 to 3-38
 - XFPL_LOG 3-32
 - XFPL_LOGFILE 3-32
 - XFPL_LOGICAL 1-4
 - XFPL_SESS_INFO 3-33 to 3-35
 - XFPL_SINGLELOGFILE 3-33
 - specifying location with
 - XFPL_INIPATH 3-18 to 3-20
 - updating free pool after modifying (OpenVMS) 3-13
 - using multiple 3-17
- xfPoolException class 8-46
- xfPool.properties file. *See* pooling properties file
- xfServer
 - using to access data remotely from
 - xfServerPlus 3-49 to 3-52
 - using to access remote SMC files 2-45, 2-48
- xfServerPlus
 - acting as client to xfServer 3-49 to 3-52
 - API 1-30 to 1-36
 - cleanup method 1-34 to 1-36
 - creating multiple sessions 3-6, 3-9, 3-13
 - data compression 3-21
 - debugging. *See* debug mode, running xfServerPlus in
 - default port 3-2
 - deploying application 3-47
 - ELB restrictions 1-3
 - encryption 3-22 to 3-30
 - error messages 3-15, 3-41 to 3-42
 - host name. *See* host name, defining on client
 - license requirements xii
 - 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 3-40
 - remote data access 3-49 to 3-52
 - running 3-2 to 3-13
 - 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 3-11 to 3-13
 - UNIX 3-8 to 3-9
 - Windows 3-2 to 3-7
 - status 3-12, 3-15
 - stopping
 - OpenVMS 3-13
 - UNIX 3-10
 - Windows 3-7 to 3-8
 - testing 3-14
 - time-out, setting 1-31
 - XFPL_LOG subroutine 1-33
 - XFPL_REGCLEANUP subroutine 1-34 to 1-36
- xfServerPlus log 3-30 to 3-40
 - default name and location 3-32
 - deleting data from 3-32
 - error messages in 3-41 to 3-42
 - log file not created 3-32
 - making application-defined entries in 1-33
 - naming 3-32
 - setting options for 3-31 to 3-40
 - single vs. multiple files 3-33
 - specifying location for 3-32
 - turning on and off 3-32
 - writing to from Java client 8-20
 - writing to from .NET client 11-24
 - XFPL_DEBUG 3-38 to 3-40
 - XFPL_FUNC_INFO 3-35 to 3-38
 - XFPL_LOG 3-32
 - XFPL_LOGFILE 3-32
 - XFPL_SESS_INFO 3-33 to 3-35
 - XFPL_SINGLELOGFILE 3-33
- xfspl 3-5, 3-7
- xfsplst program 3-14
- xfTestnet program 12-9 to 12-10
- XML file
 - for .NET API documentation 10-27 to 10-29
 - using to import/export SMC methods 2-38 to 2-40
 - See also* dbl2xml utility; genxml utility
- xml-apis.jar 7-5

