UNIFACE Configuration Guide

UNIFACE V7.2

10115087204-01
Revision 1
Mar 1999
CFG

UNIFACE V7.2
UNIFACE Configuration Guide

Revision 1

24-hour online customer support

Pillars of Wisdom is an Internet-based support service which provides real-time access to a wealth of UNIFACE product and technical information. Features include online product documentation, technical tips and know-how, up-to-date platform availability and product fixes. You can obtain full access privileges for Pillars of Wisdom by completing an online registration form (customer license information is required) at the following address: http://uniface.pillars.compuware.com/

Your suggestions and comments about UNIFACE documentation and course material are highly valued. Please send your reactions to:

Compuware Europe B.V.
Delivery Methods & Practices
P. O. Box 12933
1100 AX Amsterdam
The Netherlands

e-mail: DM&P-Hotline@nl.compuware.com
fax: +31 (0)20 311-6213

To order UNIFACE publications, contact your UNIFACE representative.

Contents

Preface

About this guide

This guide is written for system administrators who are responsible for configuring the UNIFACE environment used within their organization. It provides detailed information about the operation of PolyServer, Application Servers, and other UNIFACE components that you may need to install and configure.

Note: Previous versions of this guide contained specific information on configuring and troubleshooting operational problems on specific platforms. To make this information more accessible to people installing UNIFACE products, product-specific information now appears in the appropriate installation guide. Platform-specific chapters remain as a service to readers who are used to looking for information in this guide, but they make reference to the appropriate installation guide.

Consult the appropriate installation guide for detailed information on installing and configuring UNIFACE products on a specific platform.

How to use this guide

Read this guide to understand how UNIFACE products work and interact with each other, their platform, and environment.

The information in this guide is organized as follows:

Chapter 1 PolyServer

Explains the use of the PolyServer to access data stored on remote hosts. It includes an introduction to the PolyServer, a comparison between distributed computing and distributed databases, networking processes and errors, and the chaining of PolyServers.

Chapter 2 UNIFACE Application Servers

Explains the use of the UNIFACE Application Servers to partition the execution of your applications across remote servers.

Chapter 3 Client and peer-to-peer messaging

Describes the configuration necessary to support the posting of asynchronous messages to client or peer component instances.

Chapter 4 Assignments for a distributed environment

Explains the assignment file settings necessary to access data and non-DBMS files on server (remote) systems. The use of Application Servers to manage the execution of application components is also described.

Chapter 5 EcoTOOLS configuration

Describes how to configure your UNIFACE servers and Web applications to work with EcoTOOLS monitoring.

Appendix A UNIFACE Server Monitor

Describes the use of the UNIFACE Server Monitor to monitor and control the status of the UNIFACE daemons and processes that are active in your network.

Appendix B Microsoft Windows initialization files

Presents a complete list of the sections, and their settings, that can be specified in the Microsoft Windows **`usys.ini`** file.

Appendix C X resources

Presents a complete list of the resources, and their classes, that can be specified in the `xdefault.txt` file.

Appendix D Compatibility codes

Presents a complete list of the three-letter compatibility codes used to indicate a supported platform in V7.2.

Appendix E *Security driver*

Describes how to implement a custom security driver to replace the default UNIFACE driver, enabling you to encrypt and decrypt the user name and password.

Related information

The following UNIFACE documentation contains information that pertains closely to this guide:

- UNIFACE online help
- *UNIFACE Reference Manual*
- *Proc Language Reference Manual*
- *Quick Reference Guide*
- *Installation Guides*
- *DBMS Driver Guides*

Conventions

This section describes the conventions used in the UNIFACE user documentation.

Keystrokes

Keys that must be pressed sequentially are shown with a space between the key names. For example, GOLD R.

Keys that must be pressed simultaneously are shown with a plus sign (+) between the key names. For example, Alt+F4.

Mouse actions

The number and usage of mouse buttons can vary, so the following conventions for mouse buttons are used:

Conventions for mouse buttons.

Name	Default position	Action
SELECT	Left	Activates a control or focuses on an object.
ADJUST	Middle	Adjusts the selected objects (if supported).
MENU	Right	Activates a pop-up menu.

UNIFACE uses the SELECT and MENU mouse buttons, exclusively. If your system supports the ADJUST button, you can use it in UNIFACE (for example, in edit boxes).

The following terms are used in UNIFACE documentation to refer to mouse actions:

Terms for mouse actions.

Term	Action
Click	Press and release a mouse button.
Double-click	Click the SELECT button twice in rapid succession.
Press	Press a mouse button without releasing it.
Drag	Press the SELECT button over an object, and move the pointer to a new position before releasing the button.
Select	Either of the following: • Click the SELECT button to give focus to, highlight, or change the mode of an object. • Press the SELECT button and move the pointer to draw a frame around one or more objects before releasing the button.

Syntax descriptions

In syntax descriptions:

- Information in `Courier` font must be provided as shown.
- Information in *italics* must be replaced by actual data.
- Braces ({}) are used to show optional information.
- Vertical bars (|) are used to separate a list of options where only one item can be entered.

For example, the following syntax description indicates that the `rollback` instruction can occur with no following arguments or with either of the two optional arguments:

`rollback` {"*dbms*" | "$*path*"}

According to this syntax description, the following forms of the `rollback` statement are allowed:

```
rollback
rollback "SYB"
rollback "$MYPATH"
```

Pictures of screens

In the generic UNIFACE documentation, unless otherwise stated, pictures of screens are based on those in a Microsoft Windows 95 environment and are current at the date of publication.

Chapter 1 PolyServer

Read this chapter if you intend to access data stored on remote hosts. If you are running a UNIFACE application that accesses only local data, this chapter can be skipped.

This chapter describes the PolyServer in detail, and covers the following topics:

- Introduction to PolyServer
- Distributed processing versus distributed databases
- Networking processes
- Chaining PolyServers
- Network errors
- Further information

1.1 Introduction to PolyServer

Networking in UNIFACE means using PolyServer, a software product that allows UNIFACE applications to transparently access multiple DBMSs on remote machines via multiple network protocols.

Fortunately, when understanding how to use networking in UNIFACE, there is very little new theory to understand. At the minimal level, the *only* difference on the client side is that the application assignment file specifies a network driver instead of a DBMS driver. For the rest, using networks in UNIFACE is almost completely transparent.

PolyServer is UNIFACE split in two

The PolyServer splits UNIFACE into two physically separate parts. These two parts can be located on the same system, but are primarily intended for use on different systems in a network.

1.1.1 How the PolyServer works

To understand how the PolyServer works, consider how a stand-alone UNIFACE application works, as illustrated in Figure 1-1. All UNIFACE modules, drivers, and DBMSs operate on a single system; there is no network.

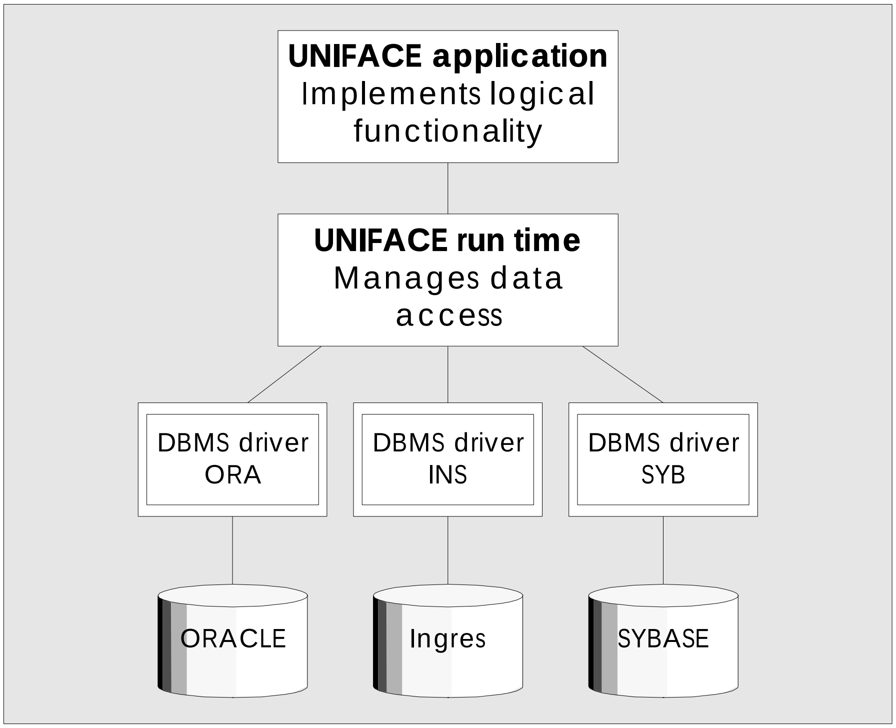


Figure 1-1 A stand-alone UNIFACE application.

The application is responsible for temporary data management. It is responsible for loading components, editing the data, checking local syntax and other constraints, and so on.

What drives the drivers?

The UNIFACE Runtime Library (URTL) handles the interaction with DBMS drivers. The URTL is responsible for formatting data before it is made available to the DBMS drivers for further handling. It is identical for every UNIFACE application. A single UNIFACE application can include drivers for several DBMSs, thus allowing the application to access multiple DBMSs simultaneously.

Remote data access

Figure 1-2 illustrates a UNIFACE application that accesses data on the client machine as well as on remote DBMSs via a PolyServer on a remote machine:

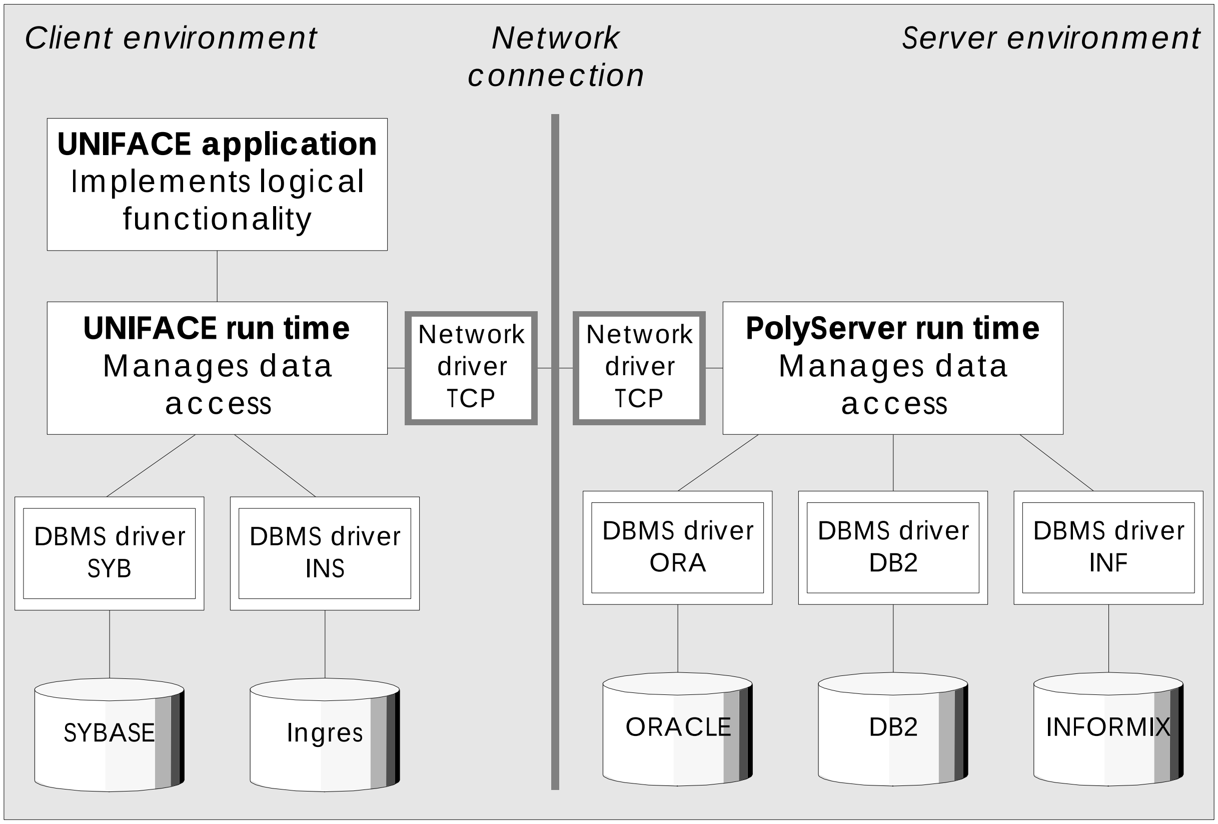


Figure 1-2 A UNIFACE application using PolyServer.

The only difference between figure 1-2 and figure 1-1 is that a network driver has been added to the client application. The network driver is responsible for handling the network software that provides access to the remote server machine, for example TCP/IP or Named Pipes.

Network driver

The network driver on the client machine is specified in an assignment file; an assignment therein directs certain entities onto a path to a network driver. An assignment can specify that data should be accessed via a network driver in the same way that an assignment specifies that the data for a particular entity is accessed via a particular DBMS driver.

For more information about assignments and assignment files, see the *UNIFACE Reference Manual*; for information about assignments and PolyServer, see section 4.1 *Assignments with PolyServer*.

A UNIFACE application can access data locally via DBMS drivers, remotely using a remote PolyServer via a network driver, or both. A PolyServer accesses DBMS data via one or more DBMS drivers and passes the data to the UNIFACE client. A PolyServer can also access another PolyServer via a network driver. See section 1.4 *Chaining PolyServers*.

A PolyServer is only active when used in combination with a client UNIFACE application. In fact, a PolyServer is only started by a request from the client to the PolyServer host. In essence, the PolyServer depends on the UNIFACE application. This is the main difference between a UNIFACE application and the PolyServer.

1.2 Distributed processing versus distributed databases

It is important to realize the difference between the terms *distributed processing* and *distributed databases*. They are related, but are very different. When designing your application and network configuration, it is important to understand which features PolyServer implements.

The ‘rollback’ and ‘commit’ functions that ensure data integrity in a distributed database are DBMS functions. UNIFACE can call these operations, but the DBMS must carry them out. Therefore, distributed database support really has little to do with UNIFACE or PolyServer.

Note: The PolyServer implements distributed processing, but does not (by itself) implement distributed databases.

Distributed processing

The term ‘distributed processing’ describes splitting required processes between several machines: the client machine executes the user’s application and a central server performs DBMS operations. This distribution has been implemented using the PolyServer. UNIFACE runs on the client machine and handles all interactions with the user and actual execution of the application. The PolyServer runs on the server machine and handles all interactions with the DBMS.

Distributed processing allows users to make the best use of the hardware they have chosen. CPU-intensive operations like editing, screen makeup, constraints checking, and so on can be performed on a desktop or departmental machine, while the central database server only needs to retrieve and store data.

Distributed databases

The term ‘distributed database’ refers to splitting data across several different machines. It allows users to access and update information from several different machines which can be geographically dispersed.

1.2.1 Logical referential integrity

UNIFACE ensures data entry is correct before committing a transaction, thereby safeguarding logical referential integrity in a transaction.

UNIFACE checks the data model. For example, if the COMPANY and EMPLOYEE entities have a restricted relationship, UNIFACE ensures that the value in the foreign key of EMPLOYEE (COMP_NUMBER, for example) refers to an existing occurrence of the entity COMPANY before allowing the user to store the new values.

1.2.2 Physical referential integrity

The DBMS ensures that a transaction is either completely committed, or not at all, thereby protecting the physical referential integrity. The DBMS checks for physical referential integrity *after* UNIFACE has determined that the logical referential integrity of the transaction is valid.

For information on two-phase commit, refer to the UNIFACE online help.

1.3 Networking processes

When using an application in a distributed environment, several processes are started on various machines, and the processes exchange a great deal of information. In most cases, there is a single PolyServer process on each server machine for each UNIFACE client process.

Multiple PolyServers on one machine

It is possible to have multiple PolyServer processes on a single machine. However, this is not a trivial task, and greatly depends on the server operating system and also on the servers' processing capacity. Avoid mixing different versions of UNIFACE, as this can easily confuse client systems and make systems difficult to maintain.

1.3.1 Which processes get started

Table 1-1 illustrates what happens when a typical user interacts with a distributed application. (The example shows cautious locking. If another form of locking is used, the stage at which a lock request is sent differs. Refer to the UNIFACE online help for more information on locking strategies.)

Table 1-1 Interaction between user, UNIFACE, and the PolyServer.

User	UNIFACE application	PolyServer
1.Starts the application	• UNIFACE loads itself into memory, initializes, and activates the first form	
2.Goes to the data entry form	• Activates a data entry form	
3.Enters a profile, then ^RETRIEVE (with the default Proc code in the <Retrieve> trigger)	• Activates the network driver (that is, path) • Sends description of the entity across the network along with generic DML open table information • Sends a Select request • Sends a Fetch request • Formats the data and updates screen	• PolyServer process activates • Logs on to the network • Logs on to the DBMS • Activates the DBMS driver, which generates and issues Select command; returns the first ten hits • Gets the data from the DBMS, formats it, and removes unneeded data
4.^NEXT_OCC	• Sends a Fetch signal • Formats the data and updates the screen	• Activates the DBMS command to Fetch the next record and returns data (TRX)
5.Modifies some of the data	• Sends a Lock request	• Activates the DBMS driver to lock the occurrence
6.^STORE (with the default Proc code in the <Store> trigger)	• Sends an Update request across the network with the modified data	• Activates the DBMS driver to update the database • Commits
7.Quits the application	• Deactivates PolyServer (closes path) • Application ends	• Logs off from the DBMS • PolyServer process ends

1.3.2 What a PolyServer process does

A PolyServer process always contains at least one network driver and one DBMS driver. The PolyServer process concerns itself only with network and DBMS I/O. The tasks performed by a PolyServer process are:

- Logging on to DBMSs and networks (that is, opening paths)
- Generating DBMS and network driver requests

- Formatting data into machine-independent UNIFACE transfer format (TRX) for sending over the network
- Formatting TRX data that has come over the network into the required DBMS format
- Logging off and closing paths

When the PolyServer is not doing one of the above, it is completely inactive.

SuperServer process

Some network protocols (for example, Named Pipes under Windows NT) do not allow a client process to start a process on the server machine. The SuperServer was developed to provide support for those environments.

A SuperServer process runs constantly on the server machine. A UNIFACE client requests the SuperServer to start a PolyServer process when necessary. The SuperServer's only purpose is to start a PolyServer process and pass the communication ID (for example, the name or number of the 'pipe') to UNIFACE. After doing this, the SuperServer goes to an idle state until the next request comes from a UNIFACE client.

1.4 Chaining PolyServers

Chaining PolyServers means accessing one PolyServer from the client machine and then redirecting the traffic on the server machine to another PolyServer on another machine. As you can see in figure 1-3, it is theoretically possible to chain an infinite number of PolyServers together. The more complicated the configuration, however, the more difficult it is to maintain.

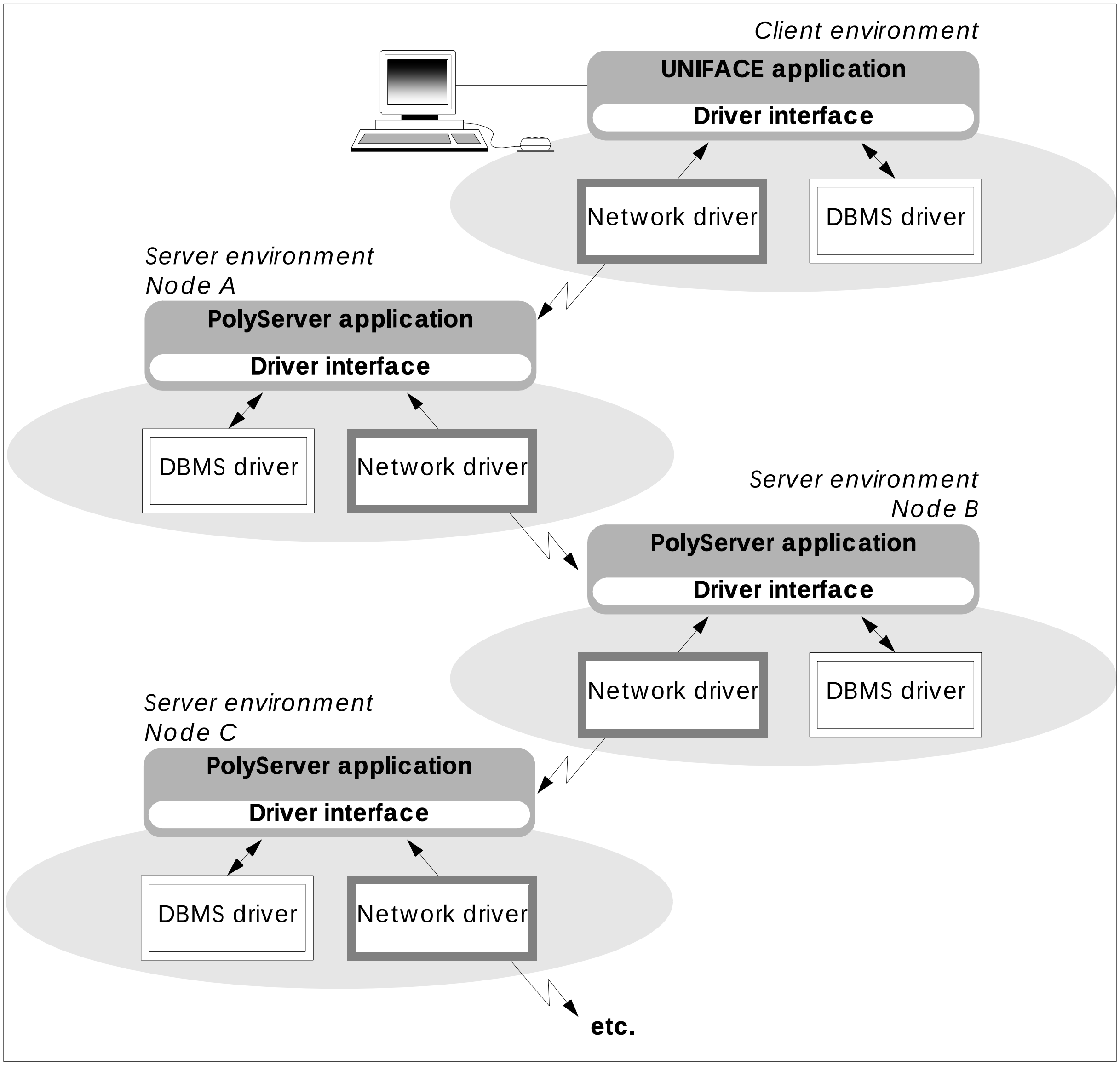


Figure 1-3 Chaining multiple PolyServers.

Assignments on each server machine can redirect traffic to another PolyServer environment. The necessary assignments can be in either **`psv.asn`** or **`PSYS:psys.asn`**.

See section 4.1 *Assignments with PolyServer* for information about which assignment files are used with PolyServer; an example of chained PolyServers with the necessary assignments is shown in section 4.1.7 *Chaining PolyServers*.

1.4.1 Why chain PolyServers?

Chaining PolyServers is a useful solution in an environment where one or more of the following conditions is true:

- You want to access a server machine over a network for which you do not have a driver, but which can be accessed if you do so first via another machine.
- You want to access a server machine over a network for which UNIFACE does not support a network driver, but which can be accessed if you do so first via another machine.
- You want to reach a database which is not available on either the client or the first server.

1.4.2 Disadvantages of chaining PolyServers

There can be disadvantages to chaining PolyServers. You should consider issues such as the following, which are discussed individually below:

- Maintenance
- Error recovery
- Passwords and logon information
- Performance

Maintenance

Chained PolyServers are difficult to maintain. The following factors are involved:

- User names, passwords, home directories, and so on
- System logon and initialization files
- Assignment files
- The different DBMSs used in the system

If you are using a PolyServer to simply redirect the network traffic, it is often easier to establish a direct link between the machines involved, unless this is impossible due to the combination of network protocols.

You should not chain two PolyServers on the same machine. It is not a problem to *run* two separate PolyServers on the same machine—simply address them separately via an assignment file.

Error recovery

If one part of a network fails, it is not currently possible to determine which part this is. Thus, it is impossible to recover from this situation. If the connection is broken, all of the subsequent connections are also broken.

Passwords and logon information

Logging on to a new machine requires logon and password information, therefore, starting up a PolyServer process on a new machine requires a fresh logon. If you have not supplied this information with a `$REMOTE_`*path* assignment, the user must supply it from the client, via the Log On form. (See section 4.1.4 *Using $REMOTE_path* for information.)

When specifying database user names and passwords for a remote server using a `$REMOTE_`*path* assignment or the Log On form, the information is encrypted before being sent over the network. This is *not* the case when using the Proc `open` statement, the `/log` command line switch, or an assignment such as `$NET=TCP:`*node*|*user*|*pwd ?*. Network passwords are never encrypted.

It is often inadvisable to let the user know about this kind of path and logon information.

Performance

Chaining PolyServers can reduce application performance to unacceptable levels. The combined performance of DBMSs, operating system, hardware, and network software can slow down an application considerably. If you are using a PolyServer simply to redirect the network traffic, it is easier to establish a direct link between the machines involved.

1.5 Network errors

This section explains what happens if the UNIFACE-to-PolyServer connection fails; in other words, communication between UNIFACE and a PolyServer has been broken for some reason. When this happens, the PolyServer cannot carry out the task requested by the UNIFACE client, so a network error occurs.

This section only explains errors that are network errors between a UNIFACE client application and a PolyServer.

What the designer must know

The designer must understand the following, which are explained in detail below:

- The principles of error handling between UNIFACE and PolyServer
- The types of network error that can occur
- How PolyServer reacts to network errors
- How UNIFACE reacts to network errors
- Network errors in a multiple PolyServer environment

1.5.1 Principles of error handling

If a network fails for any reason, the only part of the UNIFACE-to-PolyServer connection that remains is the UNIFACE client application (except in a multiple PolyServer environment). Everything else is rolled back, logged off, and shut down. This is the safest possible solution in almost all circumstances, and is therefore implemented every time.

Data protection

The way in which UNIFACE and PolyServer handle network errors is designed to protect your existing data, whatever has gone wrong. Generally speaking, the most critical time for a network error is *during* I/O, particularly when writing data to a database and committing. Similarly, data that is incorrectly retrieved can be disastrous for the correct functioning of an application.

However, network failures usually upset the environment whatever you are doing. If a network error occurs when there is no I/O, you cannot be sure that the following I/O will work as required, even if the network connection is restored in the meantime.

Context

UNIFACE maintains control blocks for each opened entity; these control blocks are sent to the PolyServer that handles that particular entity. Control blocks keep UNIFACE and PolyServer, or PolyServers, informed at all times of the current status of each opened entity; that is, they maintain the context of a running application.

Loss of context after errors

Network errors can have many effects, most of which are not good for data integrity. For example, a broken connection across a network can also cause the I/O channel to a DBMS to be broken. In other circumstances, this might not be the case. The client software can never know this, however, because the network no longer works.

You can never assume that your context has been preserved, even if the network only goes down for a few seconds and comes back up again.

1.5.2 Types of error

UNIFACE understands, and can act upon, the error categories shown in table 1-2. This table also shows the value returned to the UNIFACE application by the Proc statement that caused the error:

Table 1-2 Network errors.

Network error	$status
Unknown	-16
Pipe broken	-17
Failed to start new server	-18
Fatal	-19

These errors are explained in this section. They can be tested for in Proc code, as explained in section 1.5.3 *Testing for network errors in Proc code*.

Unknown

An ‘unknown’ error results from an incorrect (that is, unrecognized) network driver request. This can only happen when the network driver has not been correctly coded. As such, it is only likely to occur with user-defined drivers in the development stage.

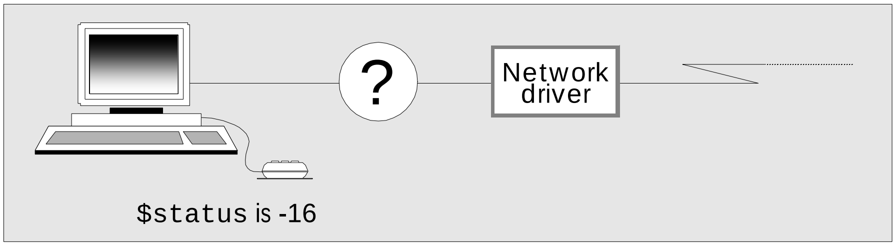


Figure 1-4 Network error: Unknown.

When an unknown network error occurs, the UNIFACE application sets `$status` and continues without interruption. In other words, UNIFACE returns control of the client application to the user and does *not* shut down the PolyServer or PolyServers.

Pipe broken

A ‘pipe’ carries the channels, or connections, over the network. A ‘broken pipe’ is therefore a network connection that has failed somewhere. This can occur between UNIFACE and a PolyServer or between two PolyServers, anywhere in the network.

When a broken pipe network error occurs, UNIFACE sets `$status` and returns control of the client application to the user. The PolyServer or PolyServers are shut down by UNIFACE. For more information about network reconnection after detection of a lost network connection, see section 1.5.5 *UNIFACE reaction to network errors*.

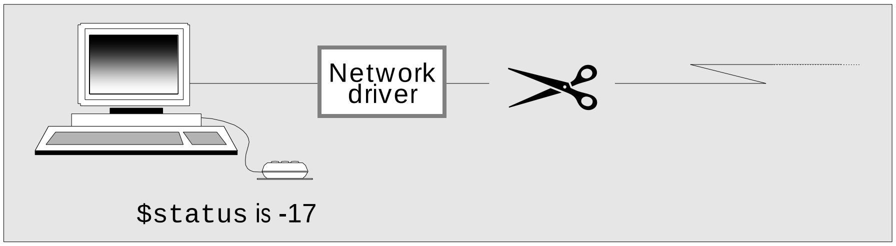


Figure 1-5 Network error: Pipe broken.

Failed to start new server

This type of network error occurs when UNIFACE or a PolyServer tries to start a new server and fails,. This can occur for any reason, for example, as a result of an incorrectly defined logon or an unplugged cable.

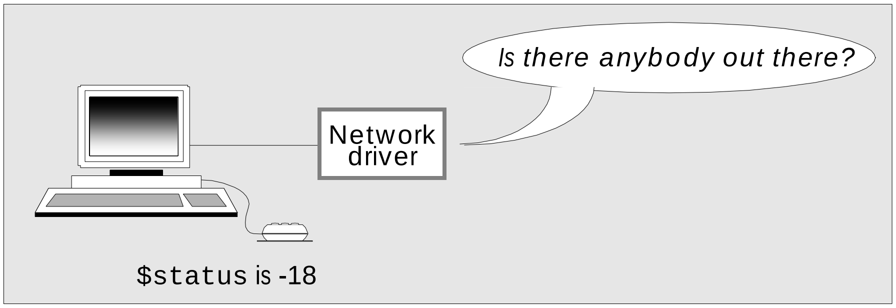


Figure 1-6 Network error: Failed to start new server.

When a ‘failed to start new server’ network error occurs, UNIFACE sets `$status` and returns control of the client application to the user.

Fatal

A fatal network error is the end of your connection with the PolyServer. This is the same as a broken pipe error, as far as UNIFACE is concerned.

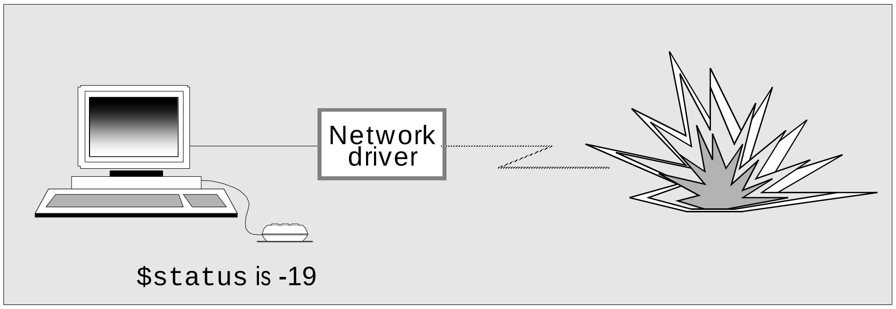


Figure 1-7 Network error: Fatal.

When a fatal network error occurs, UNIFACE sets `$status` and returns control of the client application to the user.

1.5.3 Testing for network errors in Proc code

Because UNIFACE sets `$status` to a particular value for each type of network error (shown in table 1-2), you can use these values in the Proc code in I/O triggers to test for network errors.

For example, if one entity is held in a DBMS on a remote machine, you should test in the Read trigger of that entity to see if the network is working properly when you retrieve it:

```
; trigger: Read
read
if ($status = -18)
   message "Can't connect to remote host. Contact system manager."
endif
if ($status = -17)
   message "Network connection broken. Contact system manager."
endif
```

1.5.4 PolyServer reaction to network errors

In a normally working network environment, PolyServer listens constantly to an open channel for requests from the UNIFACE application or other PolyServers in the network. When a network error occurs, PolyServer recognizes this and shuts itself down, after first rolling back any uncommitted database transactions and logging off all open DBMSs.

PolyServer issues a message before shutting down in the following situations:

- When logon information is not complete while running in batch mode
- When PolyServer attempts to return to UNIFACE but cannot because the UNIFACE process has stopped for some reason

1.5.5 UNIFACE reaction to network errors

Consider the following scenario: the client UNIFACE application is communicating over a network with a PolyServer on another machine, and the DBMS being used is on the server. The user of the client UNIFACE application enters some data. While the user is not doing anything, the network goes down. The user then stores the data.

What happens?

When the user stores, any I/O requests to the tables that require updating generate a ‘pipe broken’ error, and `$status` is set to -17. UNIFACE marks all control blocks that need the network channel in error to access the data. All subsequent I/O requests within the current transaction that need the network channel check to see if the control block is marked. If it has been marked, UNIFACE generates a fatal error and sets `$status` to -19.

Meanwhile, PolyServer recognizes that the network has gone down, rolls back all uncommitted DBMS transactions, logs off from the DBMS, and shuts itself down. The rolled back transaction can now be restarted. If the next transaction encountered has an I/O request, UNIFACE checks to see if the control block is marked. If it has been marked, it attempts to connect to a new PolyServer.

When UNIFACE succeeds in accessing the data via the network channel, it resets the mark for all control blocks.

1.5.6 Chained PolyServers and network errors

As explained in section 1.5.4 *PolyServer reaction to network errors*, the PolyServer rolls back, logs off from the DBMSs, and shuts itself down when it recognizes that the network connection has failed. This has a domino effect for each of the following PolyServers in the chain. Each PolyServer, in turn, recognizes that the connection with the previous one has been broken, and rolls back, logs off from the DBMSs, and shuts itself down. This is the only way of preserving the context of the UNIFACE application.

In figure 1-8, PolyServer 1 sends the error to the UNIFACE client. The other PolyServers shut themselves down as they realize the network has failed:

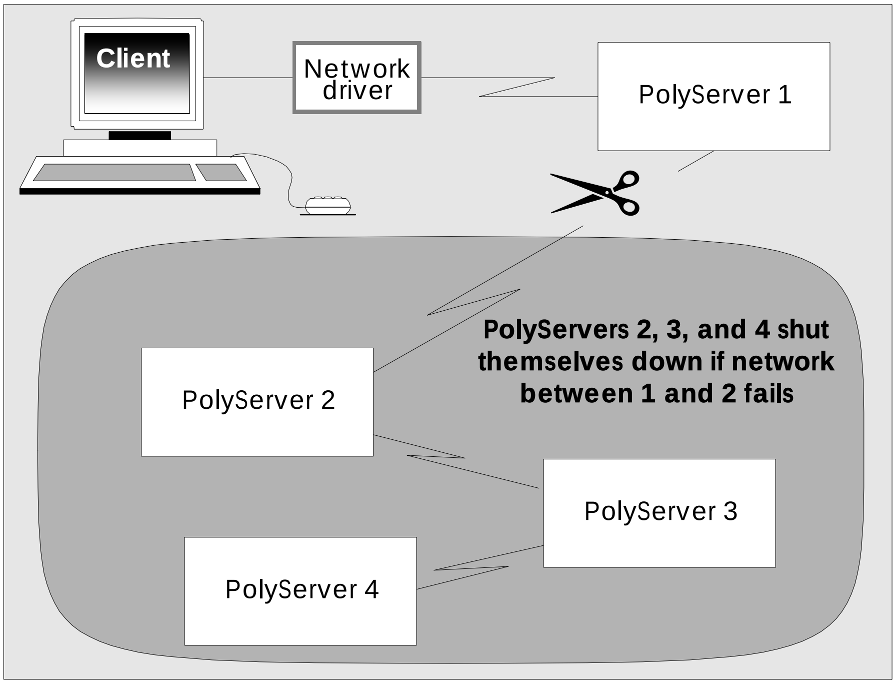

Client
Network driver
PolyServer 1
PolyServers 2, 3, and 4 shut themselves down if network between 1 and 2 fails
PolyServer 2
PolyServer 3
PolyServer 4

Figure 1-8 How PolyServers react to a chained network error.

1.6 Further information

The following sections provide useful reference information for anyone who wants to know more about the internal workings of the PolyServer.

1.6.1 PolyServer communication

UNIFACE communicates with PolyServer at the Session layer of the Open Systems Interconnection (OSI) reference model. This model is shown in figure 1-9:

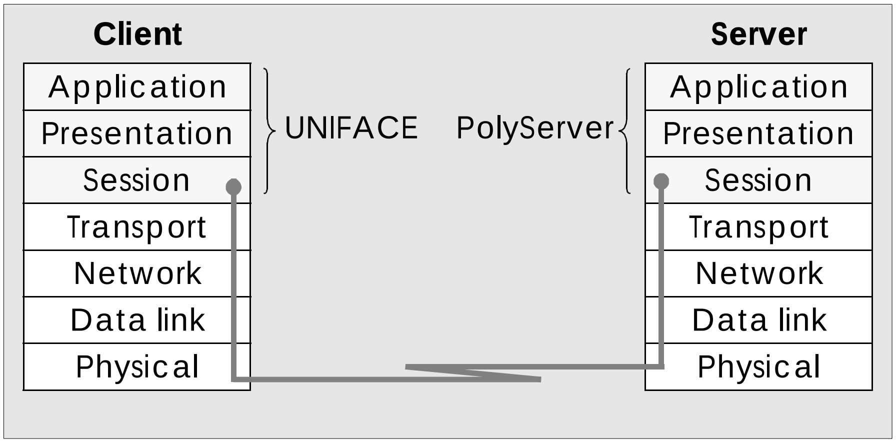


Figure 1-9 UNIFACE and PolyServer in the OSI seven-layer model.

Figure 1-9 shows that UNIFACE and PolyServer incorporate the top three levels of the OSI model. The network drivers in both UNIFACE and PolyServer each communicate with the Transport layer. UNIFACE does not concern itself with the lower layers; it is the responsibility of the networking software to provide the facilities defined below the Transport layer.

Constant communication and understanding

The two parts of the architecture, UNIFACE and PolyServer, communicate at a high and sophisticated level of understanding, each side always knowing what the other one needs and what it is doing. The interaction between these two modules is a sign of advanced intelligence and increased capacity.

This mode of operation is in contrast to many conventional client/server architectures, where the server answers a request, then forgets that request and prepares for the next one. In this situation, any subsequent requests have to be rebuilt from the beginning.

Reduced I/O

With the UNIFACE-to-PolyServer network implementation, the network carries primary information instead of lengthy command strings. Once the initial record of requirements for an application has been sent over the line, the requirements become resident on the PolyServer and can be used repeatedly. There is no need to send the full information again.

For example, if your path to a particular entity is via the PolyServer, UNIFACE sends a description of the entity to PolyServer the first time UNIFACE needs to open the entity.

Subsequent I/O activity on items in that entity requires only a very small packet of data to show which action is required on which item. This drastically reduces I/O communication time and network load overhead, therefore, the performance compares very favorably to other similar systems.

1.6.2 Network driver communication

A network driver comprises a set of routines that allow a UNIFACE client to communicate with a PolyServer, and the other way around. Each network protocol requires a separate driver. Network drivers on the client and server side are generally identical.

UNIFACE uses network drivers the same way it uses DBMS drivers in a stand-alone configuration. UNIFACE or the PolyServer activates a specific function in the network driver when it needs a service from the other process. The information needed to perform the service is available in a data structure accessible by the driver.

Data transport

A network driver is responsible only for implementing the communication between the two processes (UNIFACE and PolyServer or PolyServer and PolyServer). It is a messenger that knows nothing about the packages it transports. The network driver uses the networking facilities to do the actual data transport. Each package consists of DBMS driver function requests and responses to these requests.

The functions recognized by a network driver are described in table 1-3:

Table 1-3 Network driver functions.

Network driver function requests	Explanation
Connect	Establish a connection with the other process. (The connect routine is different between the client and server side.)
Disconnect	Drop the connection and log off.
Send	Send a package across the network. The size of each package sent is limited to 8192 bytes.
Receive	Receive a package from the other process.
Message	Interpret and format an error message received from the other process.
Information	Get information about the characteristics of the network.

Send and Receive are the most commonly used function requests. They transport all the normal DBMS function requests and responses between the UNIFACE process and the PolyServer process. The other functions set up and maintain the network connections.

Contents of network package

A network package contains the same information that is passed to a DBMS driver (that is, DBMS driver function requests and data), but in machine-independent TRX (UNIFACE transfer) format. All the standard DBMS driver requests go over the network via the network drivers.

These DBMS driver function requests are described in the UNIFACE online help, and are summarized in table 1-4:

Table 1-4 DBMS driver function requests.

DBMS driver function requests	**Explanation**
Logon path control:	
Logon	Log on to DBMS.
Logoff	Log off from DBMS.
Table control:	
Open	Open or create a database table or file.
Name	Construct the name of the overflow table or file.
Close	Close a database table or file.
Data manipulation:	
Select	Select occurrences matching a profile.
Fetch	Fetch one occurrence (from hitlist).
Write	Insert new occurrence into database.
Update	Update existing occurrence in database.
Delete	Delete existing occurrence from database.
Wildcard	Delete a set of occurrences or set their foreign key fields to NULL.
Transaction control:	
Commit	Commit transactions on path.
Rollback	Roll back transactions on path.
Miscellaneous:	
Info	Provide DBMS and driver characteristics or supply logon information.
Message	Translate an error code.
Sql	Submit a DML statement to the DBMS.

Chapter 2 UNIFACE Application Servers

Read this chapter if you intend to partition the execution of your applications across remote servers. If you are running a UNIFACE application that only executes locally, you can skip this chapter.

This chapter describes the UNIFACE Application Servers in detail, and describes the following topics:

- Introduction to Application Servers
- Synchronous and asynchronous communications
- The Component Server
- Chaining Application Servers
- Error handling

The procedure to configure the Application Servers is described in *Microsoft Windows Installation Guide* and *UNIX and MPE/iX Installation Guide*. It is recommended that you also read *Introduction to UNIFACE* if you have not already done so.

2.1 Introduction

The PolyServer, described in chapter 1 *PolyServer*, supports access to data on remote (server) systems. This effectively partitions your applications, with the application being executed on the local system, and data being retrieved on a server system.

The UNIFACE Application Server architecture extends this approach to allow you to partition the execution of your applications so that, not only can data be accessed remotely but, your applications can be executed in a distributed environment. The components of your application having no user interface (that is, services and reports) can execute on a remote server, rather than on the local system.

2.2 Synchronous and asynchronous communication

The Application Server architecture supports two modes of communication: synchronous and asynchronous. The PolyServer is an example of a synchronous server. The Application Server is an example of a server that can be executed synchronously or asynchronously. There are a number of important differences between these two modes of communication:

- Synchronous communication relies on a direct connection between the client and server systems. In asynchronous communication, a connection exists between the client and the Message Daemon for only as long as it takes to transfer the request to the Message Daemon. Once the request has been received by the Message Daemon, a local channel is opened to the Application Server and the request message is transferred.
- A synchronous server only services one client. An asynchronous server, on the other hand, can service requests from more than one client. The requested Asynchronous Server is uniquely identified by its host name, user name, and requested Server type.
- With asynchronous communication, the client does not have to wait for the request to be processed before continuing.

Limitations

Asychronous communication support is limited by the following constraints:

- Only components (services and reports) can be executed.
- It is not possible for the client system to receive a reply or return status code back from the Application Server. The only status code that is returned to the client system indicates whether the remote asynchronous request was successfully posted, not whether the requested server received the request, nor any indication about the completion status of the request.
- The asynchronous Application Server request relies on a permanently running server process, the Message Daemon, on the server system. This must be running when a request to an asynchronous Application Server is made. Normally, it is the Message Daemon that starts the requested Application Server.

2.3 The Message Daemon

The asychronous Application Server architecture relies on a permanently running process, the Message Daemon, on the remote system. The Message Daemon receives requests from any number of clients for functionality that is available on the remote system. Each client specifies the type of server required and the user account under which it is to run.

All requests on a particular host specifying the same server type and user are directed to the same Application Server. The Message Daemon administers the Application Servers running on its system.

Normally, an Application Server is started by the Message Daemon in response to a request for that server. However, it is also possible to start an Application Server manually, for diagnostic purposes. The Message Daemon handles communication to and from the client and the Application Server using Inter-Process Communication (IPC) channels.

2.4 The UNIFACE Monitor and Name Server

Two other applications are provided for use with Application Servers—the UNIFACE Monitor and the Name Server:

- The Monitor is a UNIFACE client that requests information on the servers that are running from the Message Daemon. The Monitor can also be used to stop a particular server, or to stop the Message Daemon itself. For more information on the use of the Monitor, refer to appendix A *UNIFACE Server Monitor*.
- The Name Server allows for central definitions of paths and entities through the use of a centralized assignment file. The Name Server relieves the need to have the same information repeated in each client system's assignment file. Any number of clients can use the Name Server to obtain information from a single assignment file.

2.5 The Application Server

The Application Server allows Proc statements to be executed in a distributed environment. It can be used synchronously or asynchronously. The Application Server executes the UNIFACE components on server systems. When an assignment on the client side directs an asychronous service or report to a remote server, the request is directed to the Message Daemon running on the server node.

2.6 The Component Server

The non-UNIFACE Component Server, or Component Server, extends the application partitioning of the Application Server by allowing the remote execution of 3GL services or Operating System (OS) services on remote servers. The Component Server operates in the same way as the Application Server except that 3GL code, and not Proc code, is executed remotely.

As with the Application Server, you can start the Component Server manually for diagnostic purposes. The Message Daemon handles communication to and from the client and the Component Server using Inter-Process Communication (IPC) channels.

2.7 Chaining Application and Component Servers

In the same way that PolyServers can be chained together, synchronous Application Servers can be linked together. (Asynchronous Application Servers cannot be chained together.)

Chaining Application Servers is useful where you want to use OS service functionality that is not available on either the client or the first server. It is recommended, however, that you avoid complicated chained configurations because the configuration becomes more difficult to maintain. For a more detailed discussion of chaining, refer to section 1.4 *Chaining PolyServers*.

2.8 Running and verifying the servers

This section provides an example procedure for running and verifying the servers. The procedure consists of the following steps:

1. Verify that the Message Daemon is running.
2. Start the Application Server manually.
3. Run **pdmon**.
4. Stop the Application Server using **pdmon**.

2.8.1 Verify that the Message Daemon is running

To use the asynchronous path, the Message Daemon (**umd**) must be running before any other action is taken. When the Message Daemon is running, you can test whether it is functioning properly by running the Monitor (**pdmon**) on the same node. When you run the Monitor, it uses defaults to get information from the Message Daemon. With no Application Server running, the display looks like this:

```
csh% pdmon
Monitor started, command:
Display information

UST    Username     Remote Address      State
-----  ----------   ------------------- -------------------

Monitor finished
csh%
```

This shows that the Message Daemon is running and accepting connections from its network address. The client, in this case the Monitor, has communicated successfully with the network address, and the Message Daemon has replied. If there are problems, the monitor stops temporarily and then displays an error message.

2.8.2 Start the Application Server manually

The Application Server can be started automatically or manually. In a production environment, the Application Server is usually started automatically to provide more flexibility. For diagnostic purposes, you can start the Application Server manually. This method enables you to verify network and local interprocess communication.

Note: When you start the Application Server manually, you must supply the full command line. Under Windows NT, the command line must also include the protocol and synchronicity.

If there is a problem during start-up, the Application Server exits and returns an error. Otherwise, the server creates an IPC channel, registers itself with the Message Daemon, finds its **.aps** file, and executes its application execute trigger. The Application Server then waits for input from the Message Daemon.

2.8.3 Run pdmon

If you now run **pdmon**, the Application Server should be displayed in the 'Display information' listing. Its state should be 'operational'. This verifies that the Application Server and Message Daemon can communicate over the network.

2.8.4 Stop the Application Server using pdmon

To stop an Application Server, you must identify it uniquely. Therefore, you must include `/usr` and `/ust` qualifiers in the **pdmon** command:

```
pdmon /cmd=shut /usr=user /ust=ust
```

This command sends a message to the Message Daemon. The Message Daemon then forwards the message through the IPC channel to the Application Server. If the Application Server then stops, you have verified that all servers can communicate.

Chapter 3 Client and peer-to-peer messaging

This chapter describes the configuration necessary for client and peer-to-peer messaging. It is recommended that you also read the *Proc Language Reference Manual* for information on the `postmessage` and other related Proc statements.

3.1 Introduction

With the `postmessage` Proc statement, a component instance can post an asynchronous message to either:

- A client component instance – This is described in section 3.2 *Message handling for client instances*.
- An instance activated by an independent (or peer) UNIFACE application – This is described in section 3.3 *Message handling for peer instances*.

The syntax for the `postmessage` Proc statement is as follows:

`postmessage` *Destination*, *MessageId*, *MessageData*

where *Destination* has the syntax {*InstPath*`:`}*InstName*

For more information on the use of the `postmessage` statement, see the *Proc Language Reference Manual*.

3.2 Message handling for client instances

When no *InstPath* is specified, the destination is assumed to be a client instance. That is, an instance started by a `new_instance` statement invoked by the sender. Depending on the *InstanceName* specified, the message can be routed to the following destinations, as shown in table 3-1:

Table 3-1 Message destinations for different instances

Condition	Message destination
InstanceName specifies an instance created by the client application	Instance's Asynchronous Interrupt trigger
InstanceName specifies an empty string	Sender instance's Asynchronous Interrupt trigger
InstanceName specifies an unknown instance	Local application's Asynchronous Interrupt trigger

3.3 Message handling for peer instances

When *InstPath* is specified, the destination is assumed to be an instance of an independent UNIFACE application. *InstPath* can specify either a logical path name or a physical path name of the application. These are described in the following sections.

Logical path name

In this format, *InstPath* points to a network path defined in the assignment file. For example:

```
postmessage "$MY_PATH:INST1","MSGID001","My Message"
```

where `$MY_PATH` is defined in the assignment file as:

```
TCP:myhost|paulc||UA2
```

This example assumes that a destination client application is running on the host `myhost`, under the user account `paulc`, and has registered itself to the Message Daemon (also running on the `myhost` system) with the application identification UA2.

Physical path name

In this format, *InstPath* specifies a physical network path. For example:

```
postmessage "TCP:myhost|paulc||ASV:INST1","MSGID001","My Message"
```

For both logical and physical path names, no password information is required.

Requirements

To enable peer-to-peer messaging, you must ensure the following:

- The Message Daemon is running on the host systems where the destination Application Servers or UNIFACE applications are running.
- The UNIFACE application that wants to receive external asynchronous messages is started with the UST command line switch. For example:

  ```
  UNIFACE.EXE MYAPP UST=UA1 DNP=TCP:
  ```

  The command line switches must be specified *without* a preceding slash (/) character. This registers the application with the local Message Daemon. For Application Servers, the UST command line switch is optional because the default server identification is ‘ASV’. However, for client applications, the UST command line switch must be specified.
- The client that posts messages must have a valid network path available to itself; that is, it must have registered itself with its local Message Daemon. You can use the `$instancepath` Proc function to determine how the client has been registered to its Message Daemon. For more information on Proc functions, see the *Proc Language Reference Manual*.

Broadcasting

Message broadcasting is not currently supported. If you want to send a message to a selection of clients, it is recommended that you use a subscription mechanism. Using this architecture, each interested client subscribes to a message, while the message poster application maintains a list of the clients, and posts the message to every client.

Chapter 4 Assignments for a distributed environment

Chapter 10 *Assignment files* in the *UNIFACE Reference Manual* describes the general use of assignment file, including situations in which all data and non-DBMS files are held on the client system.

This chapter describes:

- Using the PolyServer to access data and non-DBMS files on the server
- Using the Application Servers to manage the execution of application components on the server
- Using TP monitors and third-party middleware to increase transaction processing capabilities

4.1 Assignments with PolyServer

When working with the PolyServer, UNIFACE uses your assignments to determine which network driver to use and how to log on to the server. (See chapter 1 *PolyServer* for more information on the PolyServer.) Assignments on the client system can direct both DBMS entities and non-DBMS files to another node in the network.

An assignment on the client system causes UNIFACE to direct entities to the PolyServer on another node by creating a path to a network driver instead of to a DBMS driver. Assignments on the server are used to direct the entities to a DBMS driver.

In the example shown in figure 4-1, an assignment on the client for the application directs all entities on the path $DEF to the node VAX2 via TCP: (the TCP/IP driver). On node VAX2, assignments direct the entity VISITS onto the path $RDB, the entity CORRESP onto the path $SYB, and the remaining entities onto the path $RMS.

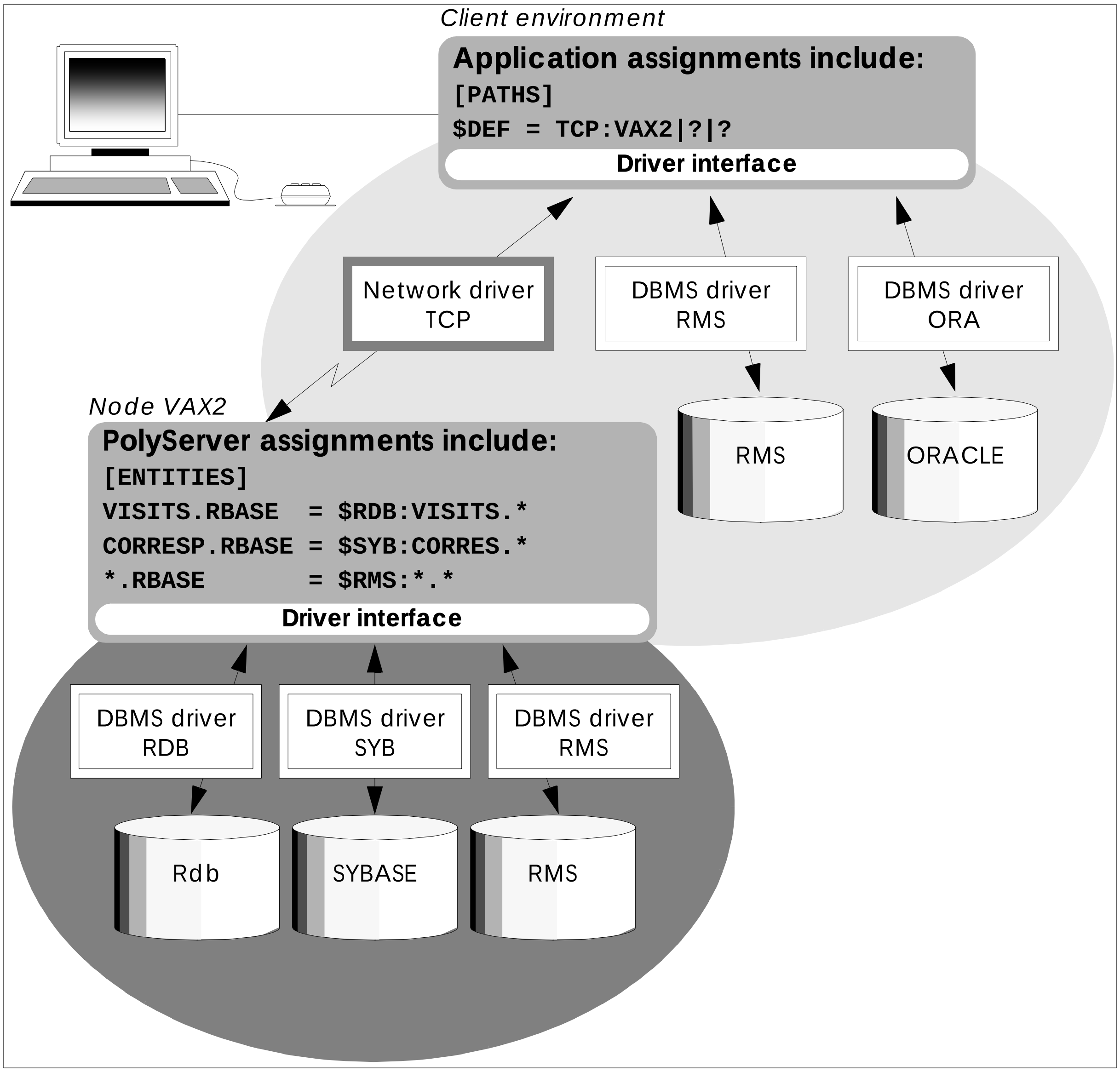


Figure 4-1   Example of assignments for the PolyServer.

4.1.1 Assignment files for the client environment

Section 10.2 *Which assignments are used?* in the *UNIFACE Reference Manual* describes the use of local and global assignment files for UNIFACE applications. Directing entities or non-DBMS files to a network path is done in exactly the same way. These objects are assigned when running a stand-alone application. The *only* difference is that the path ultimately leads to a network driver instead of to a DBMS driver; the syntax is identical.

The assignment file used by the UNIFACE application in the client environment usually does the following to direct an entity onto a network driver:

1. Creates a user-defined path that accesses the network driver.
2. Directs one or more entities to this path.

Create a path that accesses the network driver

A user-defined path-to-driver assignment optionally includes the node name, user name, and password information. If a question mark (?) is included at the end of this information, the Log On form for the network appears, asking the user for the required information.

Assign one or more entities to this path

After the path has been defined, assign the entities that are located on the server to the new path.

Example

The following assignment file contains assignments for the data used in the standard demo application delivered with UNIFACE:

```
; CLIENT.ASN
[PATHS]
$VAX3           TCP:VAX3|?|?
$VAX2           TCP:VAX2|?|?

[ENTITIES]
VISITS.RBASE    $VAX3:VISITS.*
*.RBASE         $VAX2:*.*
```

The two assignments in the [PATHS] section create paths named $VAX2 and $VAX3. Both of these paths are accessed with the TCP/IP network driver. The assignments for these paths include only the node name. The question marks (?) appearing in the position of the user name and password mean that the user will be asked for this information when needed.

The first assignment in the [ENTITIES] section directs the VISITS entity from the application model RBASE to the $VAX3 path. The next assignment directs all other entities in this application model to the $VAX2 path. Assignments on the VAX2 and VAX3 servers determine where the tables or files for the entities are found.

When the user retrieves data, the Log On form appears, requesting user name and logon information needed to access node VAX2. After logging on to VAX2, the Log On form appears again to request the user name and logon for VAX3.

The assignment file described above can be used on any client platform where the TCP/IP driver is available; the syntax does not change. The syntax is also the same when using another network driver, for example Named Pipes (NMP). The only difference is that NMP: substitutes for TCP:.

4.1.2 Assignment files for the PolyServer environment

Assignment files for PolyServer maintain a similar structure. Again, two assignment files are used: a local assignment file that contains assignments for the current PolyServer session, and a global assignment file that contains assignments for all PolyServer sessions.

The two sets of assignments are combined to make the internal assignment file for the PolyServer session. The internal assignment file for a PolyServer session is assembled in the same way as the internal assignment file for a UNIFACE application; this is described in section 10.2 *Which assignments are used?* of the *UNIFACE Reference Manual*.

Figure 4-2 demonstrates the locating of assignments for the PolyServer:

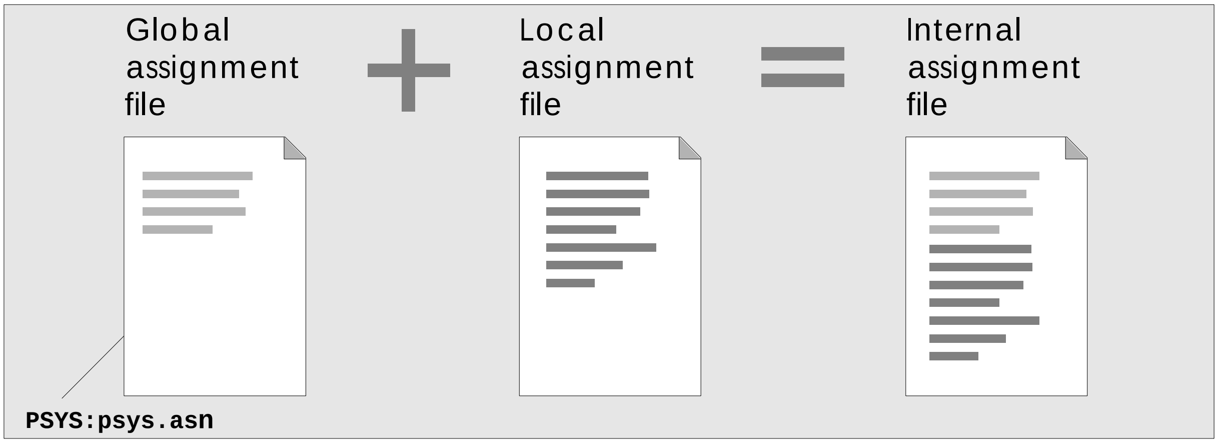


Figure 4-2 Locating assignments for the PolyServer.

Global assignment file

The global assignment file is named **`PSYS:psys.asn`** and is found in the PolyServer installation directory, **PSYS**.

Local assignment file

The local file used is the first file found that is:

- Defined with the `/asn` switch when PolyServer is started. This can be specified in the logon script in the logon directory of the server. See the *Installation Guide* for your platform for a description of how this is done.
- Named **`psv.asn`** in the user's logon directory.

Priority and scope of assignments

Bear in mind that this system gives you two assignment environments. If an assignment on the UNIFACE side directs entities to a network driver, an assignment on the PolyServer is responsible for directing these entities to a DBMS.

Generally, the assignment files on the client determine which network driver and system logon information should be used. The assignment files on the server contain DBMS assignments and logon information.

Separate hierarchies for the assignment files allow you to provide the definitions you need in the appropriate place. For example, you probably do not want end users to know DBMS passwords on the server, as this might allow unauthorized entry. Include these in an assignment on the server.

4.1.3 Relationships between assignment files

The assignments for client and server remain strictly separated from each other: the PolyServer's assignments take effect only when the PolyServer is activated by data reaching the server from the client. For example, an assignment on the server machine might reassign a $*path* that has come from the client to another $*path*, and no assignment on the client side can override this.

Figure 4-3 shows how the various assignment files work together:

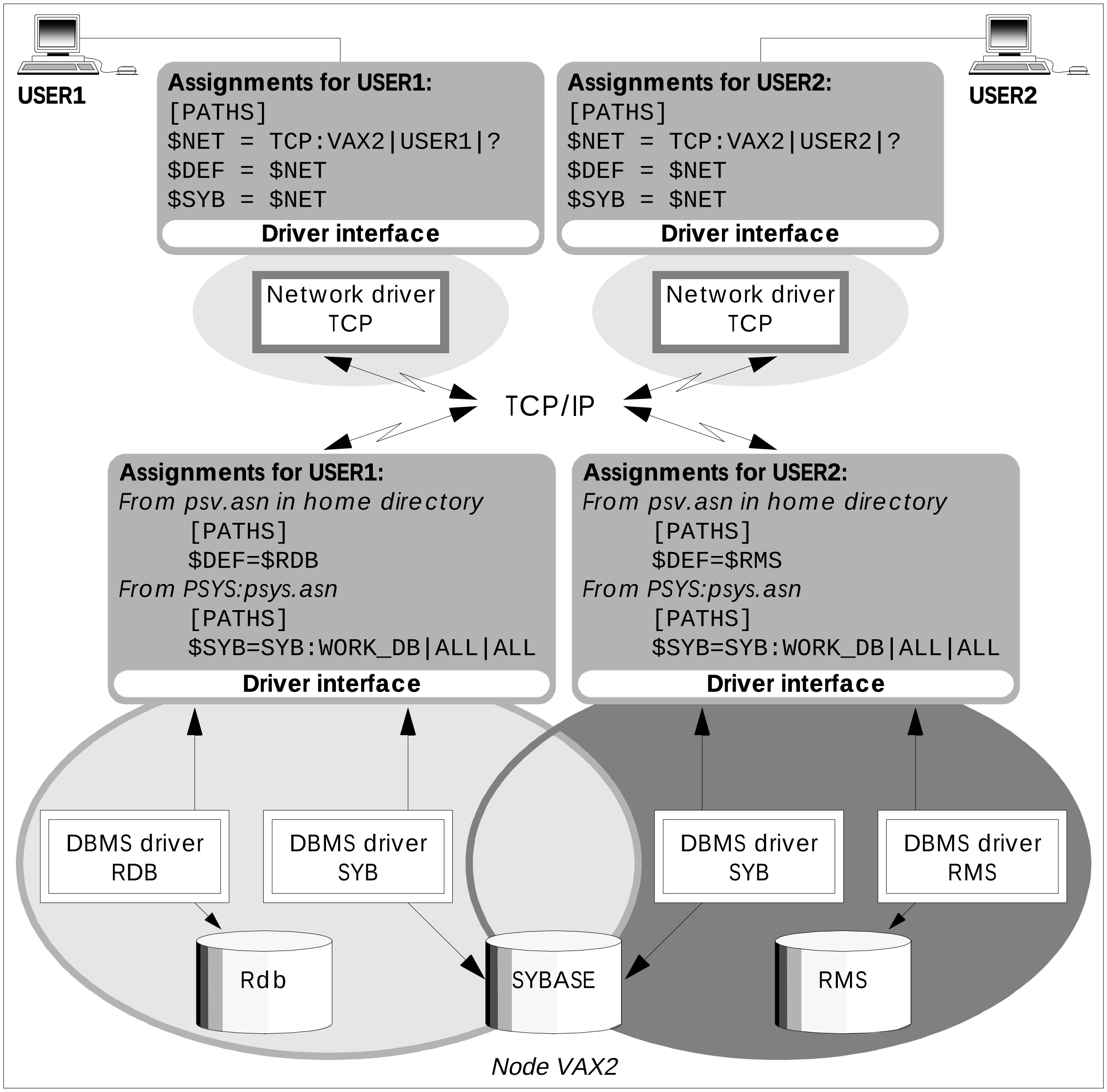


Figure 4-3 Combining assignment files with PolyServer.

In figure 4-3, two client machines are shown accessing a single server machine, VAX2. USER1's first assignment defines a path called $NET, which leads to the TCP: network driver (TCP/IP) and logs on to the node VAX2 as USER1. The next assignment directs all entities on the paths $DEF and $SYB onto the new path. USER2's assignments also direct all entities on the paths $DEF and $SYB to the TCP: network driver, but logs on to the node VAX2 as USER2.

On the server, each user's home directory contains a local assignment file **psv.asn**. In addition, there is a global assignment file (**psys.asn**) in the PolyServer installation directory (**PSYS**).

For USER1, **psv.asn** in the server environment directs all entities on the path $DEF onto $RDB, which leads to the DBMS driver for Rdb. The assignment in the global assignment file directs all entities on the path $SYB to the DBMS driver for SYBASE.

For USER2, the local assignment directs all entities on the path $DEF onto the path $RMS, which leads to the DBMS driver for RMS. The global assignment directs all entities on the path $SYB to the DBMS driver for SYBASE, using the same database as USER1.

When each client first accesses an entity on the path $NET, because of the question mark (?) in the driver logon assignment, the Network Log On form is displayed to allow entry of that user's network password.

Scope of assignment file definitions

Definitions in an assignment file are generally valid for the current machine only; this means that the assignments have a *local scope*. The assignments in any individual assignment file provide definitions for the UNIFACE or PolyServer process on the local machine only.

When you direct a path, or an individual entity, to a network driver, the assignment means that the data is located 'over there' on the network. The assignment file on the server then takes over and defines the exact location of the data, including file locations, passwords, and so on.

4.1.4 Using $REMOTE_*path*

When a PolyServer process needs to log on to a DBMS or another machine, and does not already know the logon information, it sends a request to the client. The client first looks in its internal assignment file for an assignment setting beginning with `$REMOTE_` and ending with the requested path name.

If this assignment is *found*, the logon information provided on the assignment is used to log on to the requested DBMS or network driver.

If this assignment is *not* found, the client displays a Log On form so that the user can provide the required logon information for the DBMS or network. The server becomes the ‘master’ until the information is provided. Note: this is one of very few situations in which the PolyServer asks the UNIFACE client for information.

The `$REMOTE_`*path* assignment should appear like this:

`$REMOTE_`*path* {=} *driver*`:`{*name*}`|`*user*`|`*password*

The *driver* parameter is the three-letter mnemonic for the driver. While it is not used by either UNIFACE or the PolyServer, it must be present to indicate where the database or node name begins. The logon information provided in `$REMOTE_`*path* must be complete; you cannot use question marks (?) to request a Log On form.

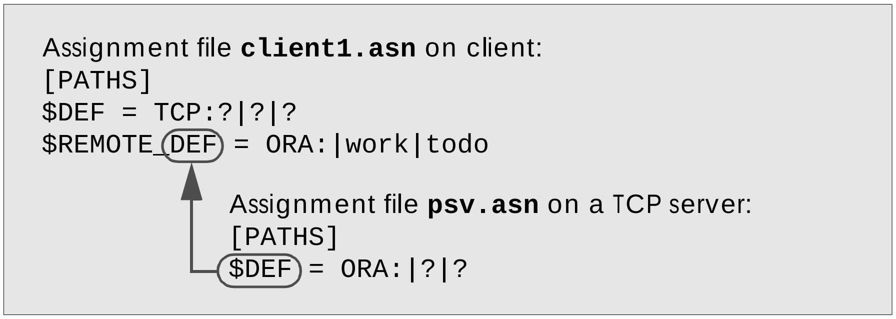
Note: The `$REMOTE_`*path assignment should appear in the assignment file section [PATHS].*

When the PolyServer sends a request for logon information, the path and driver names are included in the request. The path name used is the first path name defined for the entity. Even if that path has been assigned to another, the *first* path is sent to the client in the request for logon information.

Using $REMOTE_*path* with a default path

The example in figure 4-4 shows the assignment file **`client1.asn`** on the client, and **`psv.asn`** in the logon directory of the server. This figure illustrates a simple case where the default path $DEF is redirected to a server via the TCP driver.

When the client application first references an entity on the path $DEF, it finds an assignment in **`client1.asn`** that directs all entities on the path $DEF to the network driver TCP:. A Network Log On form appears so that the user can log on to a TCP/IP node.



Assignment file **`client1.asn`** on client:

```
[PATHS]
$DEF = TCP:?|?|?
$REMOTE_DEF = ORA:|work|todo
```

Assignment file **`psv.asn`** on a TCP server:

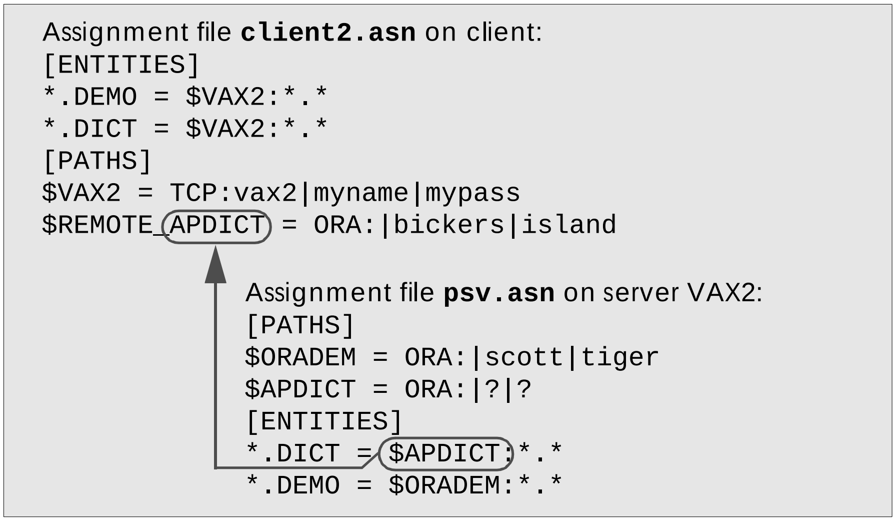
```
[PATHS]
$DEF = ORA:|?|?
```

Figure 4-4 Example using $REMOTE_*path* to provide logon information for default path.

On the server side, the assignment in **`psv.asn`** directs the entity on the path $DEF to the ORACLE driver. Since this path-to-driver assignment contains question marks (?), as opposed to actual logon information, the PolyServer asks the client for this information for the path $DEF. The client machine looks first for information from a `$REMOTE_DEF` assignment. On finding this, it returns the information to the server, which then logs on to ORACLE as user ‘work’ with password ‘todo’.

Using $REMOTE_*path* with a user-defined path

In the example in figure 4-5, the assignment files are **`client2.asn`** on the client, and **`psv.asn`** in the logon directory of the server, VAX2. When the client application first references an entity that matches the wildcard assignment *.DICT (for example, UCSCH.DICT), assignments in **`client2.asn`** direct that entity onto the user-defined path $VAX2, which leads to the TCP/IP node VAX2. Logon information is provided in the path-to-driver assignment.


Assignment file **`client2.asn`** on client:

```
[ENTITIES]
*.DEMO = $VAX2:*.*
*.DICT = $VAX2:*.*
[PATHS]
$VAX2 = TCP:vax2|myname|mypass
$REMOTE_APDICT = ORA:|bickers|island
```

Assignment file **`psv.asn`** on server VAX2:

```
[PATHS]
$ORADEM = ORA:|scott|tiger
$APDICT = ORA:|?|?
[ENTITIES]
*.DICT = $APDICT:*.*
*.DEMO = $ORADEM:*.*
```

Figure 4-5 Example using $REMOTE_*path* with a user-defined path.

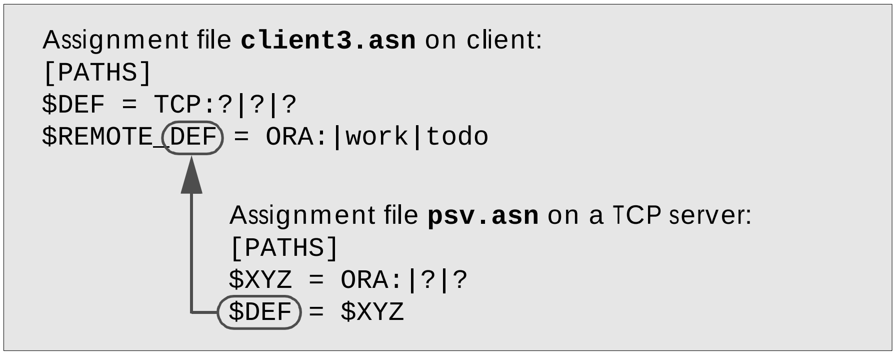
On the server, an assignment in **`psv.asn`**, directs *.DICT entities to the user-defined path $APDICT. As the definition for $APDICT uses question marks for the user name and password, PolyServer sends a request for the $APDICT logon information back to the client machine.

The client locates the required information in the `$REMOTE_APDICT` assignment and returns it to PolyServer. The server then logs on to ORACLE as user ‘bickers’ with password ‘island’.

An entity that matches the *.DEMO assignment on the client is also directed to the TCP/IP node VAX2 via the user-defined path $VAX2. On the server, these entities are assigned to the user-defined path $ORADEM. This path logs on to ORACLE as user ‘scott’ with password ‘tiger’.

Using $REMOTE_path with path-to-path assignments

The example in figure 4-6 shows the assignment files **client3.asn** on the client, and **psv.asn** in the logon directory of the server. When the client application first references an entity on the path $DEF, it finds an assignment in **client3.asn** that directs all entities on the path $DEF to the network driver TCP:. A Network Log On form appears so that the user can log on to a TCP node.



Assignment file **client3.asn** on client:

```
[PATHS]
$DEF = TCP:?|?|?
$REMOTE_DEF = ORA:|work|todo
```

Assignment file **psv.asn** on a TCP server:

```
[PATHS]
$XYZ = ORA:|?|?
$DEF = $XYZ
```

Figure 4-6 Example using $REMOTE_*path* with a path-to-path assignment.

On the server side, an assignment in **psv.asn** directs the entity on the path $DEF to a user-defined path, $XYZ. The path $XYZ is then directed to the ORACLE driver. Since this path-to-driver assignment contains questions marks, PolyServer asks the client for the logon information for the *first* path on which the entity was found, that is, $DEF. This means that the client looks for logon information in a $REMOTE_DEF setting. If this is not found, the DBMS Log On form appears for ORACLE.

Incorrect use of $REMOTE_path

You must provide complete logon information with the $REMOTE_*path* assignment. Using a question mark (?) to request a Log On form is not supported. The following example from an assignment file on the client is incorrect because it makes PolyServer try to log on using a question mark (?) as the user's password:

```
[PATHS]
$VAX2 = TCP:vax2|myname|mypass
$REMOTE_APDICT = ORA:|bickers|?
[ENTITIES]
*.DEMO = $VAX2:*.*
*.DICT = $VAX2:*.*
```

4.1.5 Assigning non-DBMS files on the network

In a PolyServer environment, it is often useful to keep non-DBMS files on the server rather than on the client. These must be directed to the network using assignments on the client side. Assignments on the server side determine the ultimate location of the non-DBMS files.

For example, the following assignment file causes the UNIFACE application to find its forms on the VAX2 machine:

```
[PATHS]
$VAX2           = TCP:VAX2|myname|mypass

[ENTITIES]
*.DEMO          = $VAX2:*.*
*.DICT          = $VAX2:*.*

[FILES]
USYS:*.FRM      = $VAX2:USYS:*.frm
*.FRM           = $VAX2:*.frm
```

If the path specified in a non-DBMS file assignment does not lead to a network driver, it is ignored.

Note: A file selection box is not able to show files on a remote machine.

4.1.6 Opening a network path using Proc code

When you explicitly open a path with the Proc `open` statement, UNIFACE assumes that you want to open a DBMS. If that path is directed to a network driver, the logon information is passed to the server for use there. If you want to provide logon information for the network driver, you must use the `/net` switch, both in the `open` statement and in the assignments for that path.

Consider the following assignments on the client:

```
[PATHS]
$SALES/NET    $TCP
$FINANCE/NET  $TCP
$UD1          $SALES/NET
$UD2          $SALES/NET
$UD3          $FINANCE/NET
```

This assignment file is used by an application in which the <Detail> trigger of a command button contains the following code:

```
open "SALESNODE|%%$$thisuser|%%$$pwd_on_net1", "$SALES/NET"
open "FINANCENODE|%%$$thisuser|%%$$pwd_on_net2", $FINANCE/NET"
open "|%%$$thisuser|%%$$pwd_on_dbms1", "$UD1"
open "|%%$$thisuser|%%$$pwd_on_dbms2", "$UD2"
open "|%%$$thisuser|%%$$pwd_on_dbms3", "$UD3"
```

When the first `open` statement is executed, UNIFACE passes the logon information for SALESNODE to the path $SALES/NET. The information is then passed to the network driver TCP:.

The next `open` statement passes the logon information for FINANCENODE to the TCP: driver along the path $FINANCE/NET. The `/net` switch in the `open` statement and the assignment statements ensures that UNIFACE recognizes that the logon information is meant for the network driver.

The next group of `open` statements passes logon information to the DBMS drivers on paths $UD1, $UD2, and $UD3. Since the `/net` switch is not present, UNIFACE assumes that this information is meant for a DBMS driver. It is passed through the network driver to the DBMS driver on the server side.

The following example is in relation to the Proc statement `open`:

```
open "node+12000|username|password|SYNC", "SYNC/net"
open "node+13013|username|password|ASYNC", "$ASYNC/net"
```

Note: It is not possible to specify a UNIFACE server type (such as the Application Server) at the end of an open statement.

For more information on the `open` statement, see the *Proc Language Reference Manual.*

4.1.7 Chaining PolyServers

Chaining PolyServers means accessing one PolyServer from the client machine, then redirecting the traffic on the server to another PolyServer on another machine. Assignments on each server can redirect traffic to another PolyServer environment. The necessary assignments may be in either **psv.asn** or **PSYS:psys.asn**.

When you are redirecting a path, your assignment is very simple. For example, the following assignment in **psv.asn** on the server machine redirects all I/O that uses the $ORA path to the TCP/IP network driver:

```
$ORA = TCP:NODE3|NODEUSER|NODEPASS
```

In this way, all data intended for ORACLE is redirected to the TCP/IP driver, which accesses a PolyServer on another machine.

Example with chained PolyServers

Consider an example where you want to run the standard demo application (with application model RBASE) in a distributed environment.

In dBase III+, the entities CORRESP and VISITS are located on your local machine. The remaining entities can be accessed following $DEF to $VAX and $VAX to node VAX1. On that node, all the entities, except INVOICE, are located in RMS. The entity INVOICE is found on the node SUN, in ORACLE. The assignments that might be used to define this situation are shown in figure 4-7:

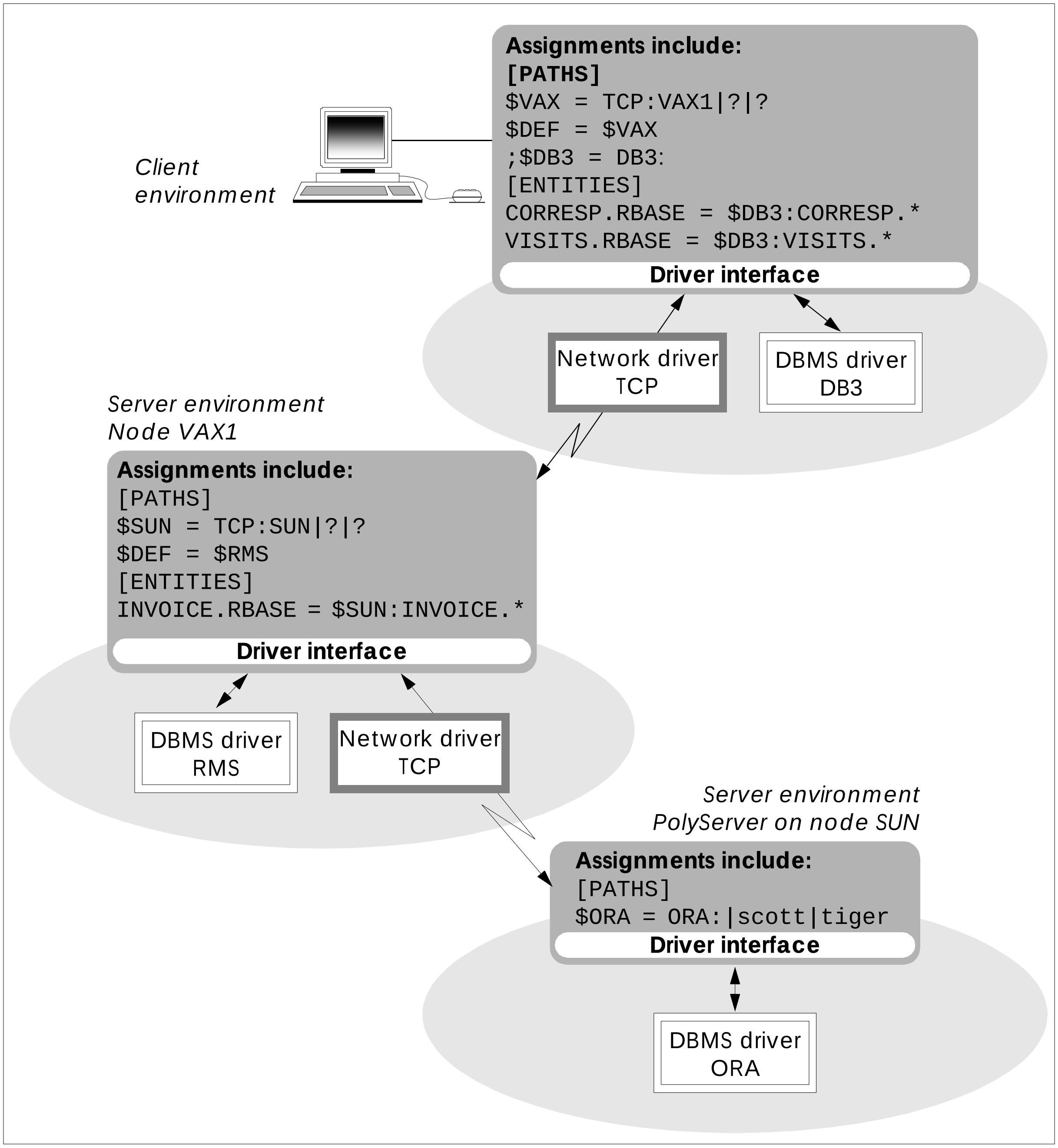


Figure 4-7 Example of chaining PolyServers.

4.2 Assignments for a distributed environment

In the simplest situation, a UNIFACE application shell and its components (forms, services, and reports) are all executed on the client. In addition, all DBMS tables and files, and all non-DBMS files (for example, the compiled component files, **`.frm`**, **`.svc`**, and **`.rpt`**), are located on the client. If any of these are found anywhere except the default locations, an assignment file can be used to direct them to another location; see section 10.4 *When no assignment file is used* in the *UNIFACE Reference Manual*.

Figure 4-8 shows a UNIFACE application on the client:

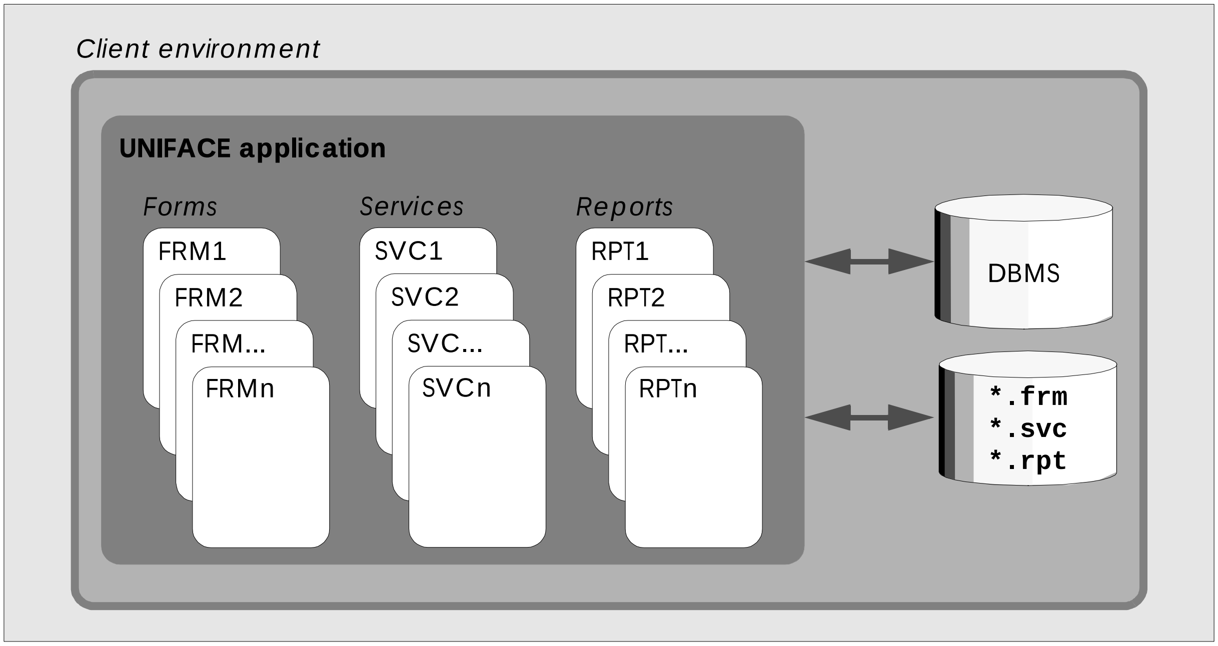
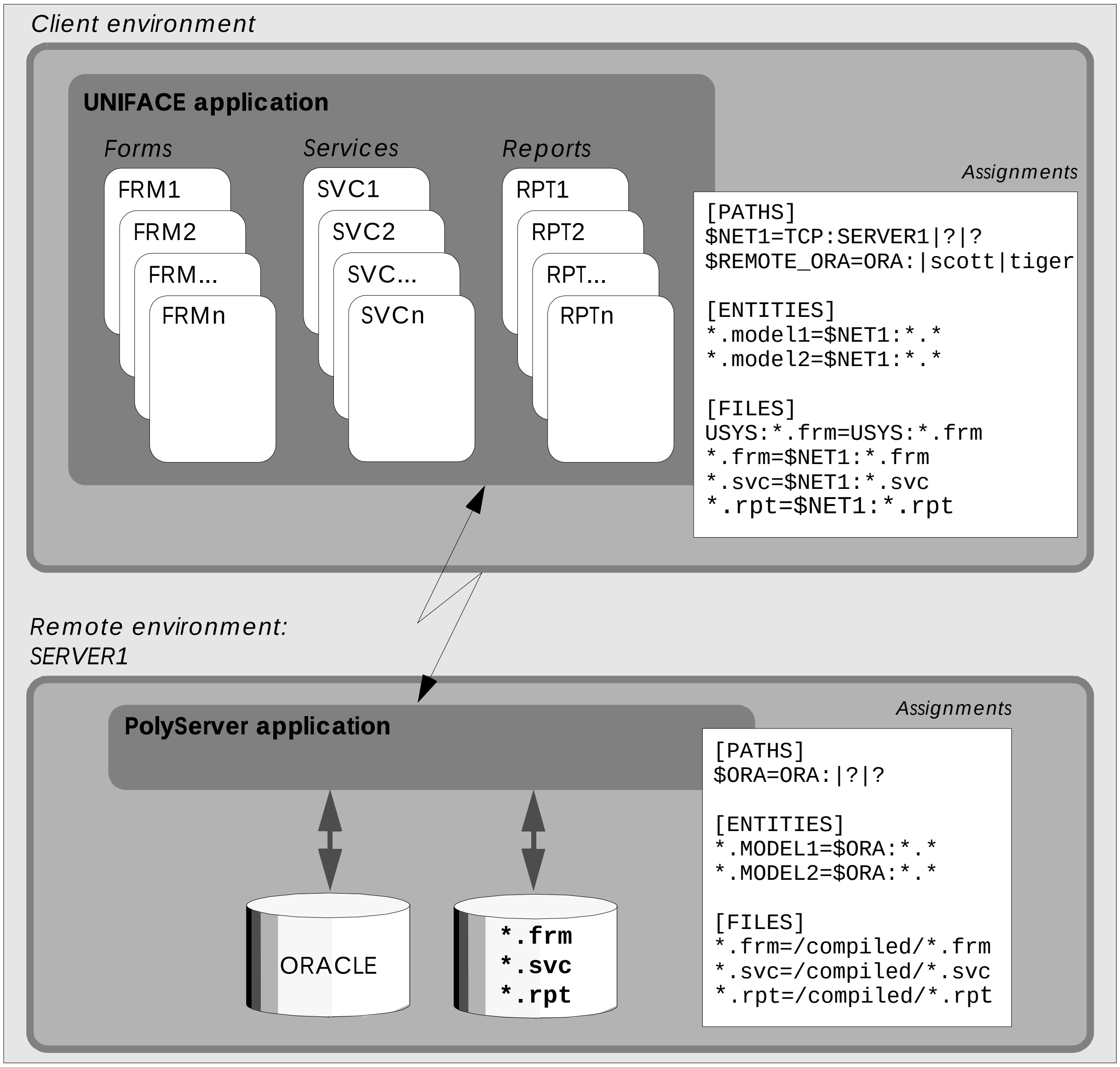


Figure 4-8 A UNIFACE application on the client.

In a more typical situation, however, the DBMS tables and files and, perhaps, the non-DBMS files are available on a server rather than on the client. An assignment file is required to direct these to the proper location. (The required assignments are described in section 4.1 *Assignments with PolyServer*.) In this case, the execution still occurs on the local machine.

Figure 4-9 shows an example of a UNIFACE application using PolyServer to access data and files on a server:



```
[PATHS]
$NET1=TCP:SERVER1|?|?
$REMOTE_ORA=ORA:|scott|tiger

[ENTITIES]
*.model1=$NET1:*.*
*.model2=$NET1:*.*

[FILES]
USYS:*.frm=USYS:*.frm
*.frm=$NET1:*.frm
*.svc=$NET1:*.svc
*.rpt=$NET1:*.rpt
```

```
[PATHS]
$ORA=ORA:|?|?

[ENTITIES]
*.MODEL1=$ORA:*.*
*.MODEL2=$ORA:*.*

[FILES]
*.frm=/compiled/*.frm
*.svc=/compiled/*.svc
*.rpt=/compiled/*.rpt
```

Figure 4-9 A UNIFACE application using PolyServer to access data and files on a server.

Partitioning your application

Since UNIFACE Seven, you have the option of ‘partitioning’ the execution of your application, allowing it to run in a distributed environment. This means that the components of your application (that is, services and reports) that have no user interface can execute on a remote server, rather than on the local machine.

The client application shell and all the forms that handle user interaction run on the local machine; assignments on the local machine direct service and report components to remote machines for execution. On each remote server, a UNIFACE Application Server manages the (synchronous) services and reports running there. See figure 4-10.

Note: Service and report components that are to be executed on remote machines must be started using the Proc instruction `new_instance` *(or* `activate`*, which does an implicit* `new_instance`*). See the Proc Language Reference Manual for further information on these instructions.*

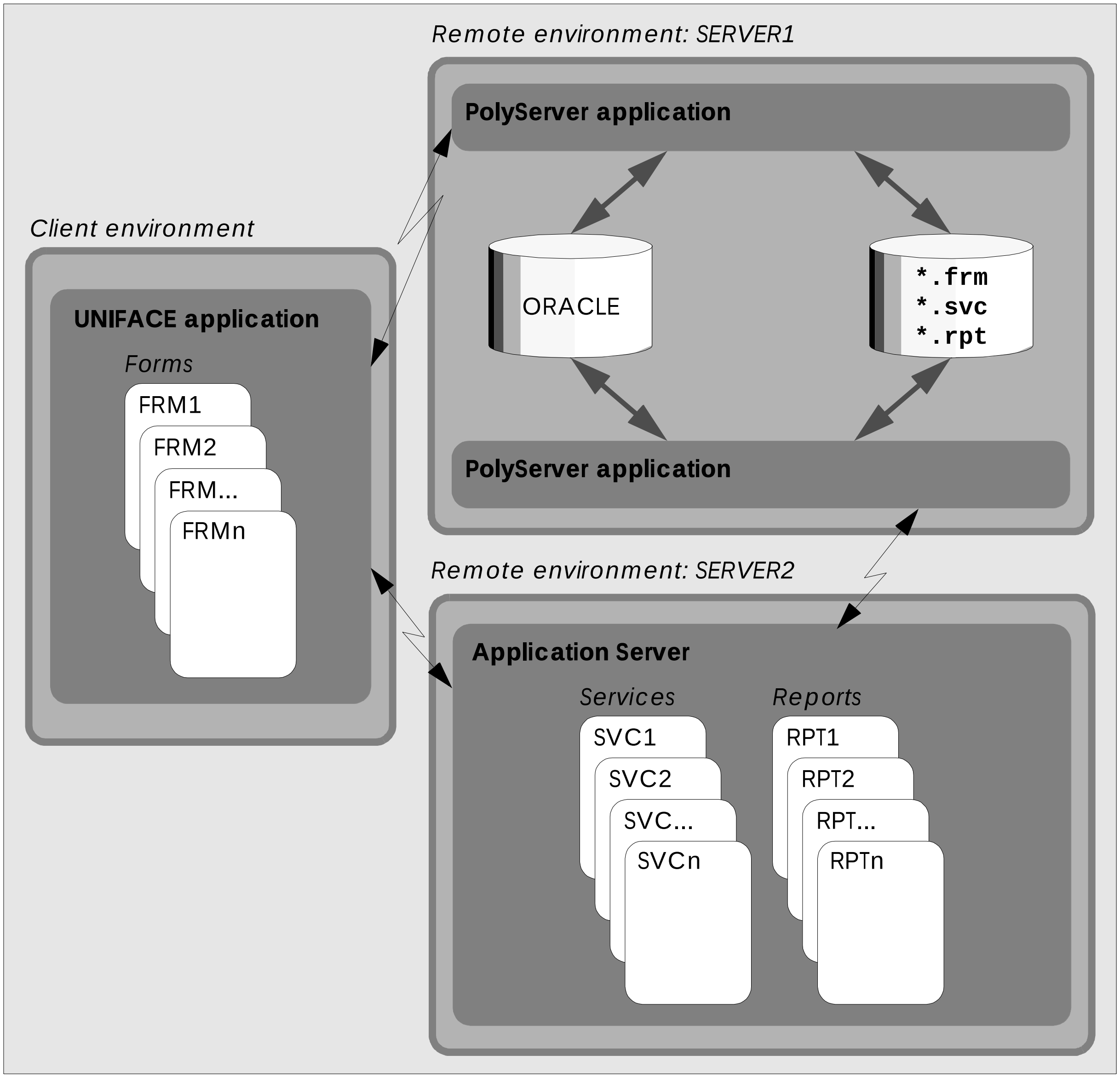
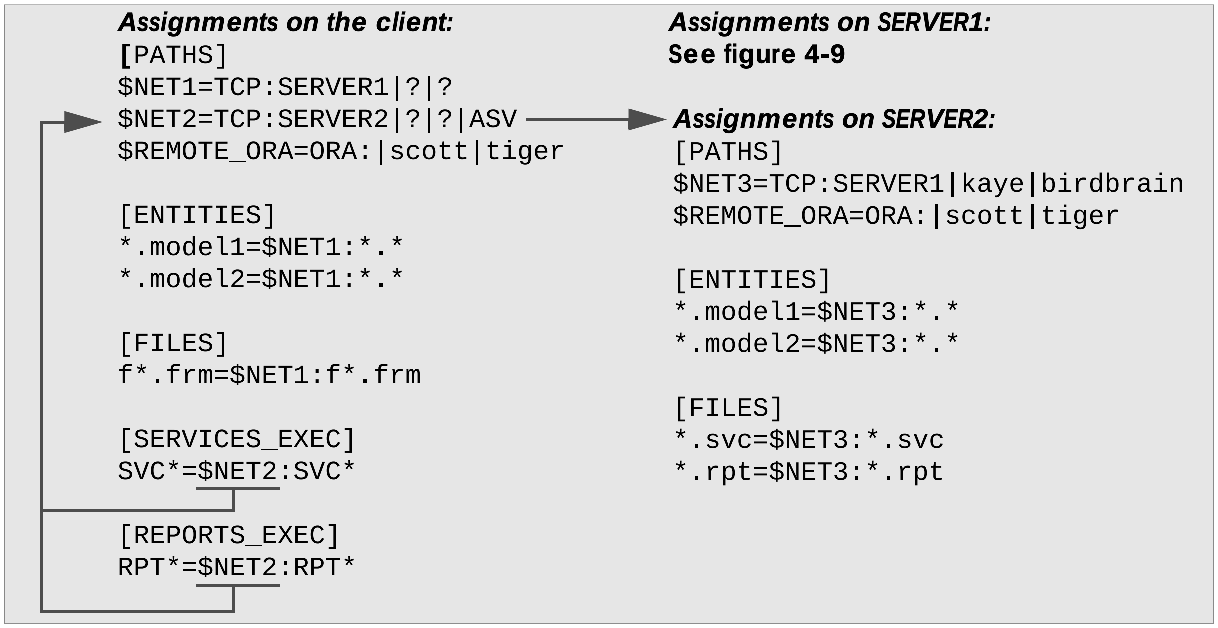


Figure 4-10 UNIFACE application on the client and services and reports on servers.

The UNIFACE application and all its forms are executing on the client. When a form running in the application starts a service or report, that component will run on SERVER2.

Figure 4-11 shows assignments that might be used to define this partitioned environment:



Assignments on the client:

```
[PATHS]
$NET1=TCP:SERVER1|?|?
$NET2=TCP:SERVER2|?|?|ASV
$REMOTE_ORA=ORA:|scott|tiger

[ENTITIES]
*.model1=$NET1:*.*
*.model2=$NET1:*.*

[FILES]
f*.frm=$NET1:f*.frm

[SERVICES_EXEC]
SVC*=$NET2:SVC*

[REPORTS_EXEC]
RPT*=$NET2:RPT*
```

Assignments on SERVER1:
See figure 4-9

Assignments on SERVER2:

```
[PATHS]
$NET3=TCP:SERVER1|kaye|birdbrain
$REMOTE_ORA=ORA:|scott|tiger

[ENTITIES]
*.model1=$NET3:*.*
*.model2=$NET3:*.*

[FILES]
*.svc=$NET3:*.svc
*.rpt=$NET3:*.rpt
```

Figure 4-11   Example assignments in a distributed environment.

On the client, assignments in the sections [SERVICES_EXEC] and [REPORTS_EXEC] direct the service and reports components to the network path $NET2; these components will be executed on the server defined by the path-to-driver assignment for $NET2. This path opens a TCP connection to SERVER2, and uses the 'ASV' symbol to start an Application Server on that node. (See section 4.2 *Assignments for a distributed environment*.)

The path-to-driver assignment for $NET1 leads to a PolyServer process that manages access to data and non-DBMS files. When the path is first opened, the PolyServer process is started on the remote node, SERVER1. In the same way, the first time that a service or report on path $NET2 is referenced, an Application Server process is started on the remote node SERVER2. This process then manages all remote components that execute on this node.

On the server side, assignments in the [ENTITIES] and [FILES] sections direct entities and non-DBMS files to SERVER1. See section 4.2.2 *Assignments on the Application Server side.*

Asynchronous communication

When an assignment on the client side directs an asynchronous service or report to a remote server, the request is directed to the UNIFACE Message Daemon running on the server node. The Message Daemon manages asynchronous communication between the network driver and the UNIFACE Application Servers for all client UNIFACE applications using that server.

Each client application starts a UNIFACE Application Server process on the server where it starts a synchronous service or report. The UNIFACE Message Daemon, however, manages the asynchronous services and reports for *all* client applications accessing that server. The Message Daemon starts an additional Application Server for each network user who activates an asynchronous service or report, and for each symbol that is used to define the path.

For example, if two clients, USER1 and USER2, are starting synchronous and asynchronous services and reports on node SERV1, and both log on to the network as user ‘general’ with symbol ASV, different Application Servers are created on SERV1:

- For the synchronous services and reports of USER1
- For the synchronous services and reports of USER2
- For the asynchronous services and reports of network user ‘general’ using the symbol ASV

This situation is illustrated in figure 4-12:

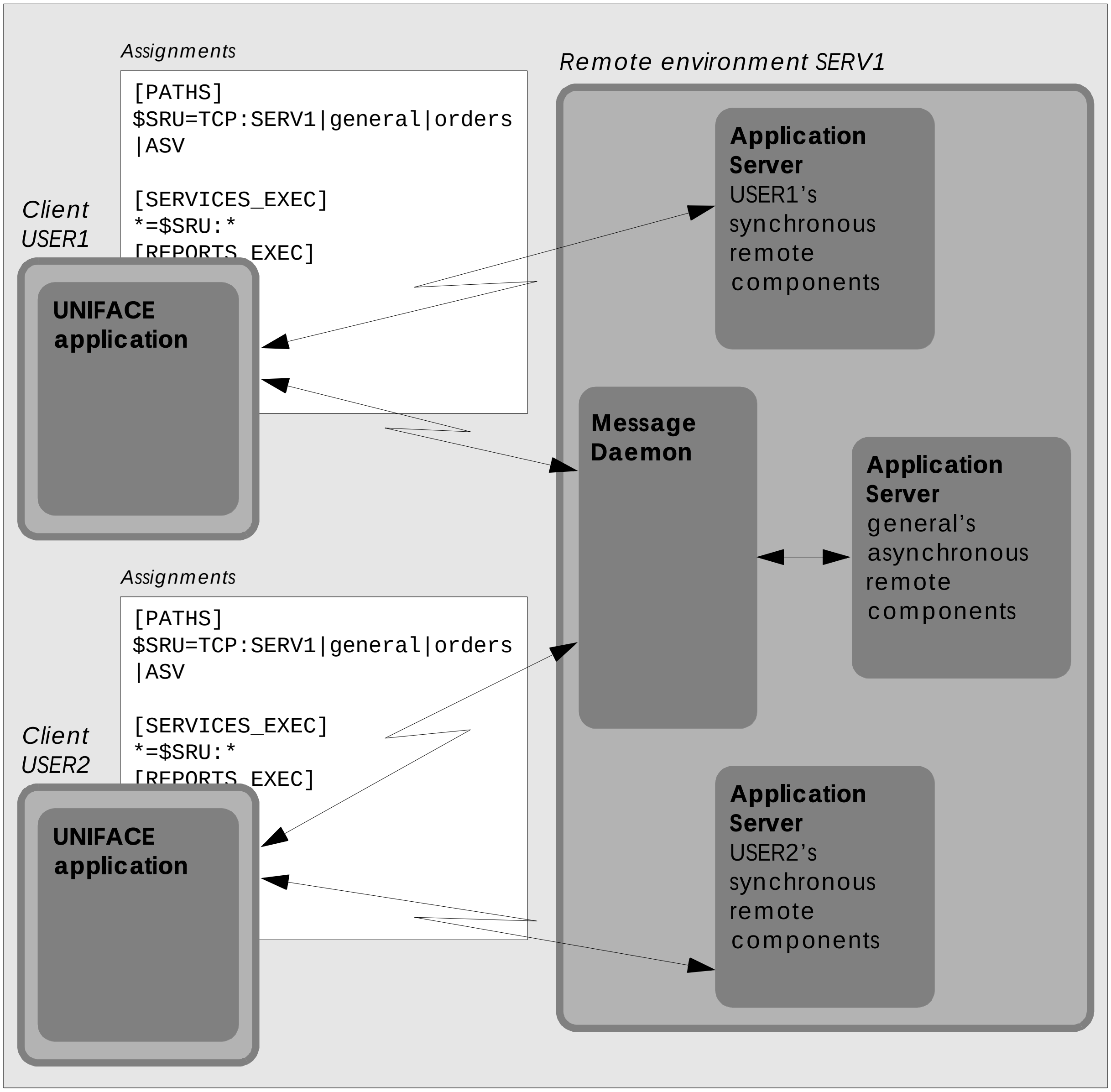

Assignments
[PATHS]
$SRU=TCP:SERV1|general|orders
|ASV
[SERVICES_EXEC]
=$SRU:
[REPORTS_EXEC]
Client
USER1
UNIFACE application
Assignments
[PATHS]
$SRU=TCP:SERV1|general|orders
|ASV
[SERVICES_EXEC]
=$SRU:
[REPORTS_EXEC]
Client
USER2
UNIFACE application
Remote environment SERV1
Application Server USER1's synchronous remote components
Message Daemon
Application Server general's asynchronous remote components
Application Server USER2's synchronous remote components

Figure 4-12 A server environment with a Message Daemon.

If clients USER1 and USER2 use different user names to log on to the network (for example, USER1 logs on as user 'general', and USER2 logs on as 'major'), the Message Daemon creates separate Application Servers to handle their asynchronous components. See figure 4-13:

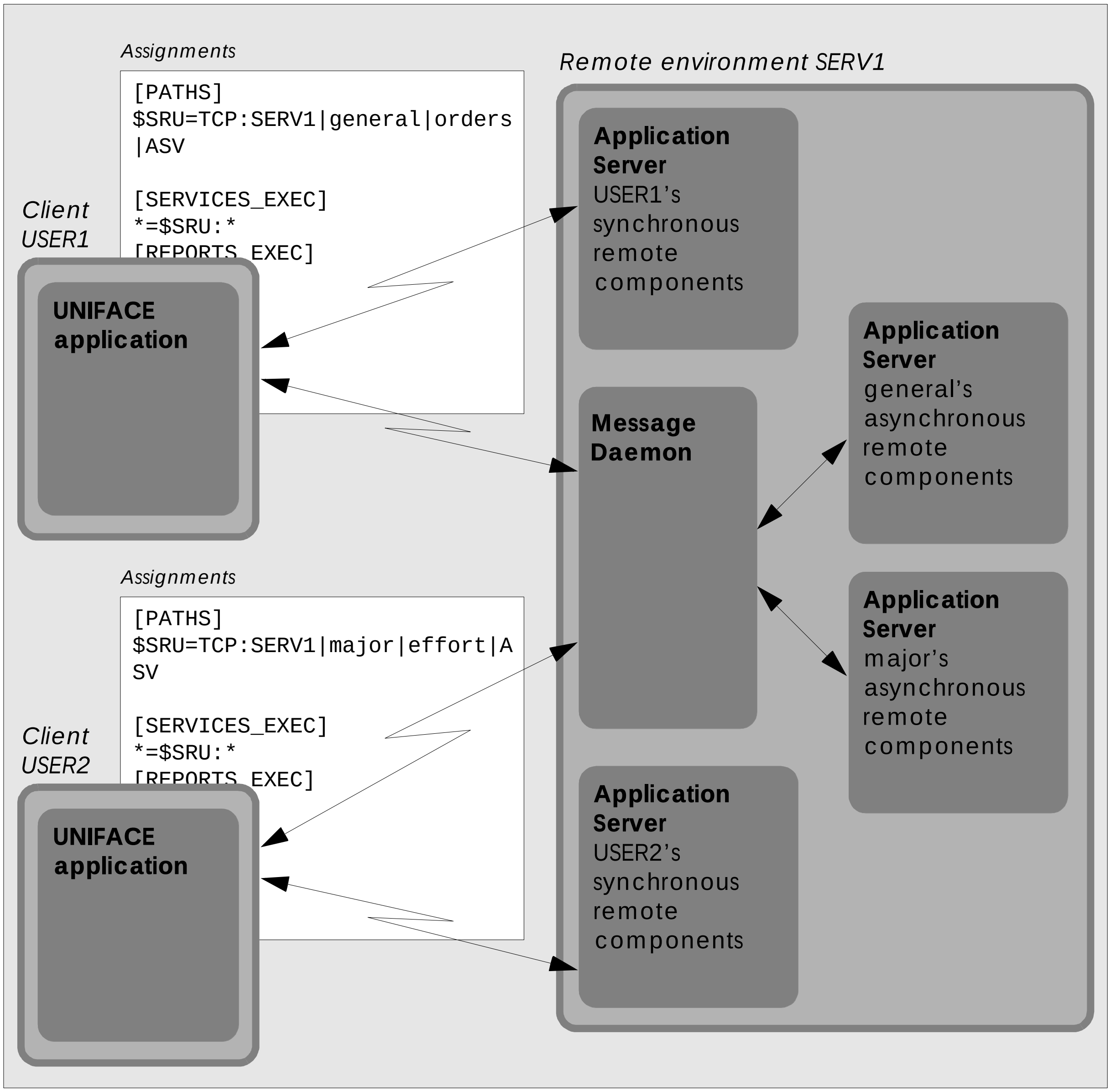

Assignments
Remote environment SERV1
[PATHS]
$SRU=TCP:SERV1|general|orders|ASV
[SERVICES_EXEC]
=$SRU:
[REPORTS_EXEC]
Client USER1
UNIFACE application
Application Server USER1's synchronous remote components
Message Daemon
Application Server general's asynchronous remote components
Application Server major's asynchronous remote components
Assignments
[PATHS]
$SRU=TCP:SERV1|major|effort|ASV
[SERVICES_EXEC]
=$SRU:
[REPORTS_EXEC]
Client USER2
UNIFACE application
Application Server USER2's synchronous remote components

Figure 4-13 The Message Daemon creates separate Application Servers for different users.

Note: The Monitor allows you to monitor and control the status of the UNIFACE Message Daemons and asynchronous Application Servers that are active in your network. See appendix A UNIFACE Server Monitor for more information.

Message frame

All information that is directed to the message frame of a synchronous Application Server automatically appears in the client's message frame.

All information that is directed to the message frame of an asynchronous Application Server is lost, unless the assignment setting `$PUTMESS_LOGFILE` is used to direct it to a file.

For more information, refer to the *UNIFACE Reference Manual*.

4.2.1 Assignments on the client side

The assignment file used by the UNIFACE application on the client usually does the following to direct a service or report component to an Application Server on a remote server:

- Creates a path to a network driver using user-defined paths.
- Directs one or more services or reports to this path.

Note: See chapter 4 Command line switches in the UNIFACE Reference Manual for information on using the command line subswitch `/asv` *used with* `/app` *or* `/lin` *to create an Application Server. See the UNIFACE online help for details on the requirements for an Application Server on the platforms where it can be used.*

Path to network driver

A path to a network driver can be created with a path-to-driver assignment (see section 10.8 *Path-to-driver assignments* in the *UNIFACE Reference Manual*) or with user-defined paths (see section 10.10 *User-defined paths* in the *UNIFACE Reference Manual*). In either case, the final 'step' in the path to the driver is a path-to-driver assignment.

In general, path-to-driver assignments appear in the section [PATHS] and are used to provide logon information for DBMS and network drivers that require this information. For creating a partitioned application environment, the syntax for this assignment has been extended with an extra argument that indicates the type of server to be found on the path:

`$path` {=} *driver*:{*name*}|{*username*}|{*password*}{|*server_type*}

Where:

- *driver*, *name*, *username*, and *password* arguments are described in section 10.8 *Path-to-driver assignments* in the *UNIFACE Reference Manual*.
- *server_type* is one of the following:
  - PSV, indicating that $path leads to a PolyServer. If *server_type* is omitted, this is the default.
  - UNS, indicating that $path leads to a UNIFACE Name Server. (See section 4.3 *Using the UNIFACE Name Server* for information on the UNIFACE Name Server.)
  - A symbol defined on node *name*; typically, this is ASV. This symbol defines the command necessary to start the Application Server on node *name*.

    For example, if *name* is a UNIX platform, an environment variable should be defined in user's **`.profile`** :

    ```
    $ASV = /home/bin/asv /asn=asv.asn ust=asv tcp:
    ```

    See your platform's *Installation Guide* for more information on the requirements for starting an Application Server in your environment.

[SERVICES_EXEC] and [REPORTS_EXEC]

These sections contain two-part assignments that you can use to determine the node on which your services and reports execute.

Each assignment in these sections is of the following form:

```
component1 {=} {$path:}component2
```

You can use wildcards for assigning components. These are used in the same way as wildcards in non-DBMS file assignments; see section 10.6.1 *Wildcards in non-DBMS file assignments* in the *UNIFACE Reference Manual*.

`$path` should be a path that is defined in the [PATHS] section of your assignment file; this path should lead to the remote node that will execute the requested service:

- If the path leads to an Application Server, the service or report executes on that node.
- If the path leads to a UNIFACE Name Server, the assignment file there directs the services and reports to the node where they will be run. (See section 4.3 *Using the UNIFACE Name Server* for further information.)
- $SRU is the default path, if the path is not specified.

$SRU is a default path, created at installation, that can be used to direct UNIFACE service and report components to a remote server for execution. (See section 10.3 *Assignments after installation* in the *UNIFACE Reference Manual* for information on the default paths created at installation.)

By default, all remote components are directed to this path. If no assignment can be found for a service or report (for example, if there is no [SERVICES_EXEC] or [REPORTS_EXEC] section or if the remote component does not match any of the assignments there), UNIFACE looks for an assignment for the path $SRU. If there is no assignment that directs $SRU to an Application Server or a UNIFACE Name Server, the component executes locally.

For example, in the simplest case, where all services and reports are directed to a single server, the [SERVICES_EXEC] and [REPORTS_EXEC] sections are not required; however, the [PATHS] section must include an assignment for $SRU:

```
[PATHS]
$SRU = TCP:phoenix|kaye|birdbrain|asv
```

The [SERVICES_EXEC] and [REPORTS_EXEC] sections can also be used to direct services and reports to different Application Servers:

```
[PATHS]
$SRU1 = TCP:phoenix|kaye|birdbrain|asv
$SRU2 = TCP:roc|kaye|birdbrain|asv

[SERVICES_EXEC]
svc_*  = $SRU1:*
*      = $SRU2:*

[REPORTS_EXEC]
rpt_*  = $SRU1:*
*      = $SRU2:*
```

Default paths for OS and 3GL services

The Component Server contains a subset of the Application Server's functionality but can only execute 3GL services. The $SOS and $S3C default paths direct OS services and 3GL services, respectively, to a Component or Application Server for execution.

As with the $SRU default path, if no assignment can be found for an OS or 3GL service, UNIFACE looks for an assignment for the paths $SOS or $S3C. If there is no assignment that directs $SOS or $S3C to an Application or Component Server, the component executes locally.

4.2.2 Assignments on the Application Server side

Assignment files for a UNIFACE Application Server maintain a similar structure to a normal UNIFACE application; that is, two assignment files are used, a local assignment file and a global assignment file.

The two sets of assignments are combined to make the internal assignment file for the Application Server session. The internal assignment file for a Application Server session is assembled in the same way as the internal assignment file for a UNIFACE application; section 10.2 *Which assignments are used?* in the *UNIFACE Reference Manual* describes this procedure.

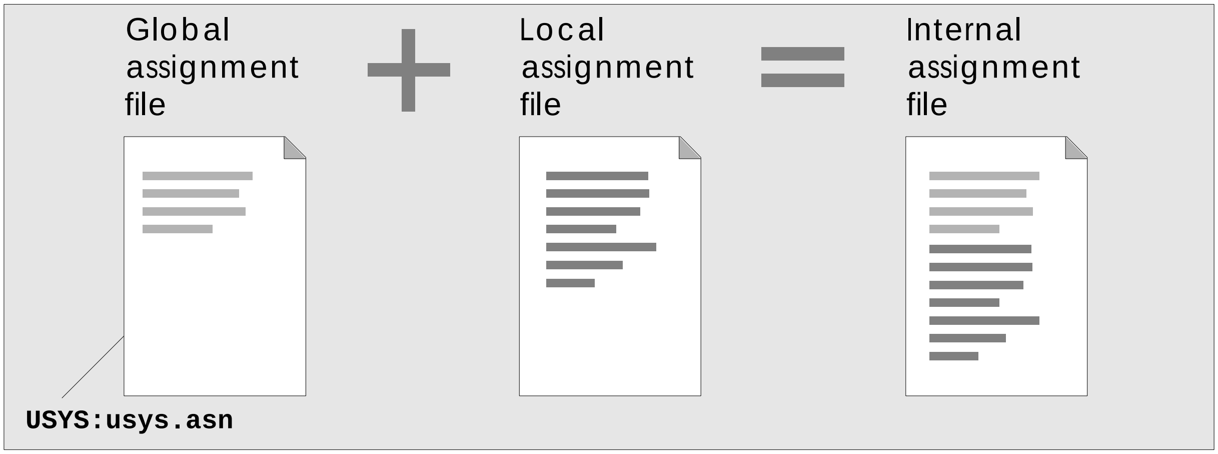


Figure 4-14 Locating assignments for the Application Server process.

Global assignments

The global assignment file, **USYS:usys.asn** is located on the server where the Application Server is running. This is the same file used by any UNIFACE application running on that machine. (See section 10.2 *Which assignments are used?* in the *UNIFACE Reference Manual*.)

Local assignments

The local assignment file used is the first file found that is:

- Defined with the /asn switch when the Application Server is started.
- Named ***appl*.asn** in the user's logon directory, where *appl* is the name of the compiled Application Server executable, created using the /asv subswitch when the application was compiled.

Priority and scope

The Application Server is a separate UNIFACE application process, so you have two assignment environments. If an assignment on the client side directs a component to a network driver, assignments on the Application Server side are responsible for directing the entities and files that the component requires to the proper location. When the same entities and files are used by the client application, this can mean that assignments are duplicated on the client and the server.

4.3 Using the UNIFACE Name Server

When many clients are running distributed applications in a networked environment, the situation can be very dynamic, making it difficult to manage. The UNIFACE Name Server allows you to simplify the management of this environment by centralizing the assignments required to locate the execution of remote components (services and reports).

The UNIFACE Name Server is a daemon that runs on one of the nodes in your network. When it receives a request from a client to locate the execution of a service or report component, it returns the required node name and allows the client to proceed with the process of logging on to that node, if required.

Figure 4-15 illustrates using the UNIFACE Name Server:

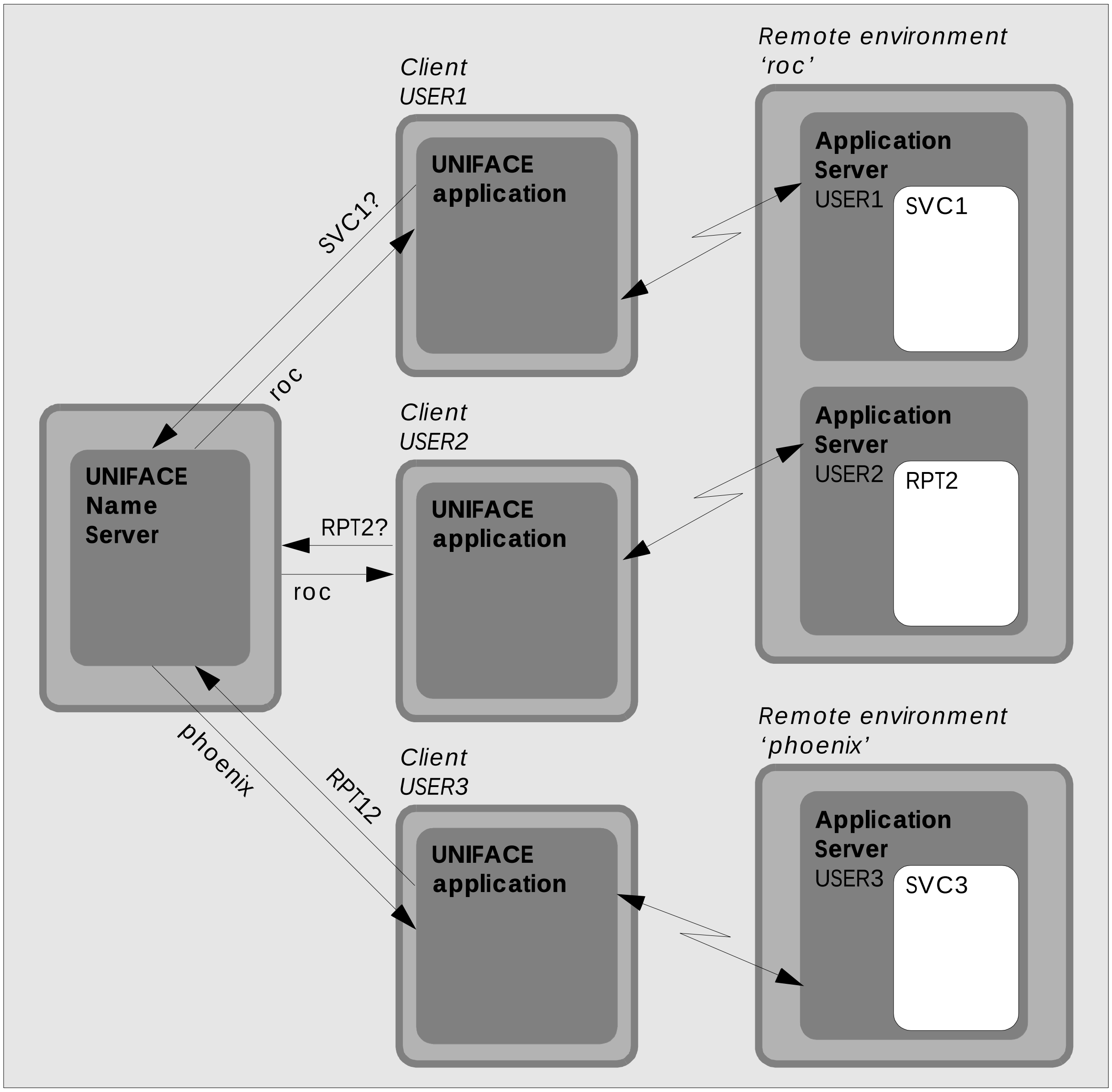


Figure 4-15 The UNIFACE Name Server instructs clients as to where their services and reports should be executed.

To make use of the UNIFACE Name Server, assignments on each client direct the execution of remote components to a path that leads to the Name Server. Usually this is done using the default path $SRU. The [SERVICES_EXEC] and [REPORTS_EXEC] sections are needed only if components need to be renamed locally, or if some components will be located without use of the Name Server (perhaps, to execute locally).

The Name Server determines where the execution of services and reports should occur. Assignments in the [SERVICES_EXEC] and [REPORTS_EXEC] sections direct the remote components to paths. Assignments in the [PATHS] section direct these paths to the appropriate host server. The server name is then returned to the client. An assignment on the client is responsible for logging on to the remote server to execute.

Note: Compuware provides a monitor program, `pdmon`, *that allows you to monitor and control the status of the UNIFACE Name Server in your network. See appendix A UNIFACE Server Monitor for more information.*

Example: Executing all remote components on servers

Consider the example shown in figure 4-16:

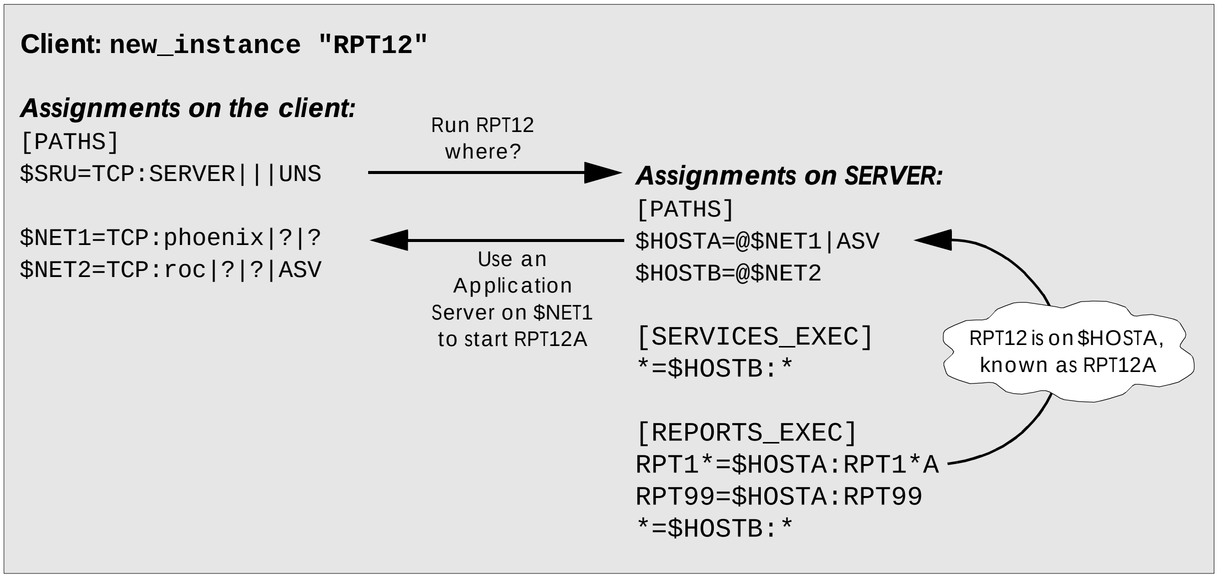


Figure 4-16 Assignments on the UNIFACE Name Server direct the remote component to a server.

The client application asks to create a new instance of report RPT12. There is no [REPORTS_EXEC] section in the client assignments, so the report follows the path $SRU. This path leads to a UNIFACE Name Server on node SERVER.

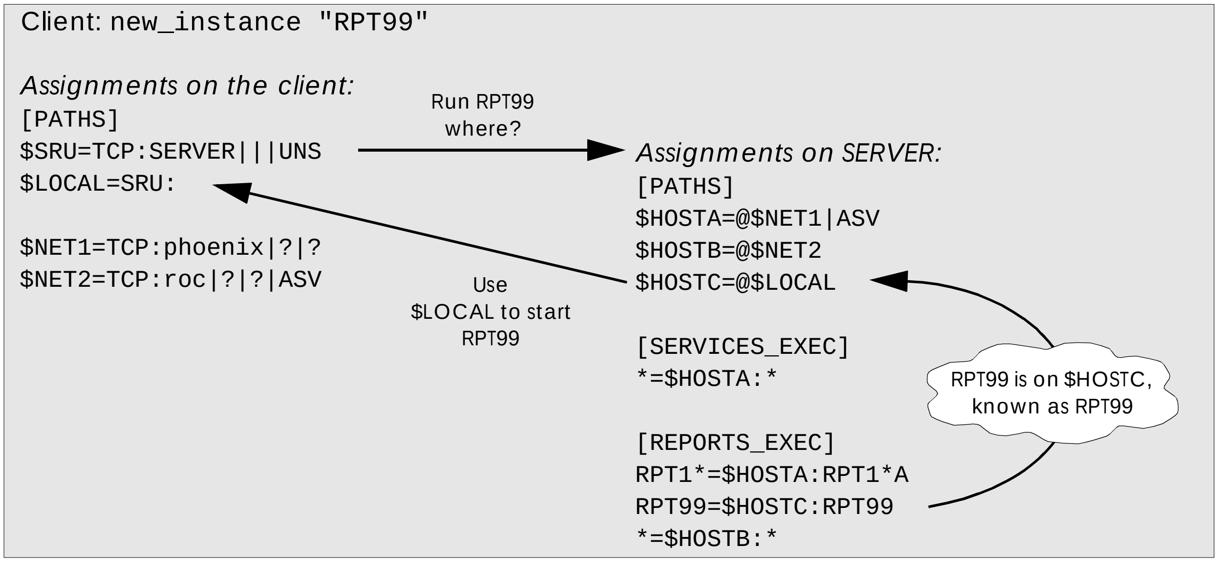
The Name Server on node SERVER then uses its assignments to locate where the remote component will be executed. In the [REPORTS_EXEC] section, RPT12 matches the first assignment; it is renamed to RPT12A and directed to path $HOSTA.

An assignment in the [PATHS] section on the Name Server directs $HOSTA to an Application Server (the server type 'ASV') on path $NET1; the at sign (@) instructs it to return to the client to open the selected path. (Note that the server type can be specified by the Name Server's assignments or by the client's assignments. If specified in both places, the client's assignment takes precedence.)

In the client assignments, a path-to-driver assignment opens the path $NET1 to node PHOENIX using the TCP network driver (if the path is not already open). Assignments at node PHOENIX must locate the compiled report file, **`RPT12A.rpt`**, as well as DBMS tables and non-DBMS files needed by the report.

Example: Executing a remote component on the client

Consider the example shown in figure 4-17:



Client: `new_instance "RPT99"`

Assignments on the client:

```
[PATHS]
$SRU=TCP:SERVER|||UNS
$LOCAL=SRU:

$NET1=TCP:phoenix|?|?
$NET2=TCP:roc|?|?|ASV
```

Assignments on SERVER:

```
[PATHS]
$HOSTA=@$NET1|ASV
$HOSTB=@$NET2
$HOSTC=@$LOCAL

[SERVICES_EXEC]
*=$HOSTA:*

[REPORTS_EXEC]
RPT1*=$HOSTA:RPT1*A
RPT99=$HOSTC:RPT99
*=$HOSTB:*
```

Figure 4-17 Using the UNIFACE Name Server to direct a remote component to the client.

The client application asks to create a new instance of report RPT99. There is no [REPORTS_EXEC] section in the client assignments, so the report follows the path $SRU. This path leads to a UNIFACE Name Server on node SERVER.

The Name Server on node SERVER uses its assignments to locate where the remote component will be executed. In the [REPORTS_EXEC] section, RPT99 matches the second assignment; the report is directed to path $HOSTC.

An assignment in the [PATHS] section on the Name Server directs $HOSTC to path $LOCAL; again, the at sign (@) tells it to return to the client to open that path. The path $LOCAL must be defined in the client's assignments.

In the client assignments, a path-to-driver assignment assigns the path $LOCAL to the driver SRU:, this 'driver' is the client application. This means that RPT99 will be executed locally by the client application.

Beware of loops!

When you are using the Name Server to locate the execution of remote components, assignments on the client direct a component to the Name Server. The Name Server, in turn, returns a path name to the client to complete the assignment. 'Looping' occurs when the client sends a component to the Name Server to get a path name; the Name Server returns a path name, and the client directs this path name back to the Name Server. In this case, the Proc instruction `new_instance` (or `activate`) returns an error.

You should avoid this from occurring when building assignment files for the client and Name Server.

4.3.1 Assignments on the client side

To direct a service or report component through the UNIFACE Name Server, the assignment file used by the UNIFACE application on the client side usually does the following:

- Creates a path to the Name Server (via a network driver) using user-defined paths (or the default path $SRU).
- Directs service and report components to a user-defined path that leads to the UNIFACE Name Server. If any remote components are not assigned, the path $SRU is used to locate where they will execute.
- Defines path-to-driver assignments for all possible paths that can be returned by the Name Server.

Path to UNIFACE Name Server

A path to a network driver can be created with a path-to-driver assignment (see section 10.8 *Path-to-driver assignments* of the *UNIFACE Reference Manual*) or with user-defined paths (see section 10.10 *User-defined paths* in the *UNIFACE Reference Manual*). In either case, the final 'step' in the path to the Name Server is a path-to-driver assignment, as described in section 4.2.1 *Assignments on the client side*. In this case, UNS is required for the *server_type*:

`$path` {=} *driver*`:`{*name*}`|||UNS`

[SERVICES_EXEC] and [REPORTS_EXEC]

The assignment file sections [SERVICES_EXEC] and [REPORTS_EXEC] are used to determine where service and report components execute. The contents of these sections are described in section 4.2.1 *Assignments on the client side.*

By default, the path leading to the node where all service and report components are executed is $SRU. If all remote components are to be located by the UNIFACE Name Server, the [SERVICES_EXEC] and [REPORTS_EXEC] sections are not needed on the client side; all services and reports can simply follow the path $SRU to the Name Server. In this case, a path-to-driver assignment (in the [PATHS] section) should direct $SRU via a network driver to the Name Server.

If some service and report components are to be executed locally, these components should be directed either through assignments on the client side, or from the Name Server to a path that is assigned to the SRU: driver. This 'driver' is actually the client application, where local services and reports will run.

Paths to all possible nodes

The [PATHS] section of each client assignment file should include path-to-driver definitions (as described in section 4.2.1 *Assignments on the client side*) for each path that can be located by the Name Server.

If the server type is not provided by the Name Server assignment, it should be provided in the client assignment. If specified in both places, the client's assignment takes precedence. If no server type is specified, it defaults to PSV, for PolyServer.

4.3.2 Assignments on the Name Server side

Assignments on the UNIFACE Name Server do the following to locate the execution of remote service and report components:

- Return the path name and server type to the client using special path-to-server assignments (optionally, in the section [PATHS])
- Direct the remote service and report components to the proper host using the assignment file sections [SERVICES_EXEC] and [REPORTS_EXEC]

Path-to-server assignments

A path-to-server assignment is used to return to the client the path name and server type where the service or report, that is currently being assigned, should be executed. An assignment that points to a network driver and logs on (if the path is not already open) must exist on the client.

The syntax for a path-to-server assignment is:

`$path` {=} @$*client_path*{|*server_type*}

Where:

- $*path* is a path that appears in the assignment file sections [SERVICES_EXEC] or [REPORTS_EXEC].
- $*client_path* is a path that is defined in the client assignments.
- *server_type* is usually a symbol defined on the node reached by the path $*client_path*. (This symbol is described in section 4.2.1 *Assignments on the client side*.) The *server_type* argument determines the type of the UNIFACE server that will execute the service or report currently being assigned.

If the *server_type* argument is omitted, UNIFACE expects to determine the type of server from the $*client_path* assignment. If the server type is also omitted on the client, PSV (PolyServer) is assumed; this is not an appropriate UNIFACE server for a service or report.

If the *server_type* argument is defined both on the Name Server and on the client, the client's assignment is used. However, it is recommended that you include the *server_type* argument on the Name Server side, since the main use of the Name Server is to centralize the assignments needed to locate the execution of remote components.

For path-to-driver assignments, you can also define alternative fallback paths to reach data on remote machines. For information on fallback paths, see the *UNIFACE Reference Manual*.

[SERVICES_EXEC] and [REPORTS_EXEC]

The assignment file sections [SERVICES_EXEC] and [REPORTS_EXEC] are used to determine where service and report components execute. The contents of these sections are described in section 4.2.1 *Assignments on the client side*.

On the Name Server, if the *$path* part of component assignment is omitted, only the name of the component being located is returned to the client; the client assignments are then used to determine where the component will be executed. This means that the client assignment file must use the sections [SERVICES_EXEC] and [REPORTS_EXEC] to direct these components to a path that leads to an Application Server. If you choose to use this technique, make sure that the component is not directed back to the Name Server, resulting in an assignment loop between the client and the Name Server.

Which assignments are used?

The UNIFACE Name Server uses only a local assignment file. It uses the first of these files which can be defined with the `/asn` switch when the Name Server daemon was started, or the file **`uns.asn`** in the start-up directory of the Name Server daemon.

4.3.3 Middleware support

UNIFACE supports the execution of components over foreign middleware and currently supports the following middleware drivers:

- TUX (TUXEDO)
- ENC (ENCINA)
- OVS (CORBA)
- COM
- CIC (CICS)
- IMD (IMS/DC functions)

The driver mnemonics listed here are used in the assignment files. For example, you can assign the $SRU path to $TUX. This changes the path of services to a TUX driver. UNIFACE also allows you to add your own middleware driver (MW1).

4.3.4 Transaction Processing

UNIFACE supports Transaction Processing (TP) management through the use of TP monitors. UNIFACE supports Encina and TUXEDO TX/XA transaction management. These monitors keep track of the transactions that occur between clients and servers. The clients and servers may be distributed over a wide area. For more information on the Transaction Processing support provided by the Universal Request Broker Architecture, see the *URB Interfaces Manual*.

Transaction Management

The details of the TP monitor are declared in the Transaction Management [TM] section of the assignment file. This section specifies:

- `MANAGER` – The name of the transaction manager used.
- `PATHS` – The database (servers only) and ‘activate’ paths (which have transaction attributes).
- `TIMEOUT` – The session time-out, in seconds, for the transaction. The default value is 0 (no time-out).
- `TXRETURNPHASE` – A user must enter numeric value 1 or 2. Entering a value of 1 means that the client will not wait for the second phase of the commit to end. Entering a value of 2 means that the client will wait for the end of the commit. The default is 2.

- `LOGON{DB path}`– This keyword registers the TP monitor to the database. For example, LOGON$ORA=.. Logon information should only be supplied for Encina

At a minimum, the [TM] section must specify a MANAGER and a PATH. If the client assignment file has a [TM] section, the server assignment file also needs to have a [TM] section. Transactional clients cannot use the server if it does not have a [TM] section.

Table 4-1 Supported paths for middleware

Path	Example
$TUX	$TUX TUX:node+port|user|password|application_password|client
$ENC	$ENC ENC:[ENCINA cell name]|user|password
$OVS	$OVS OVS:[IPAddress][+port|+][+poolname|+user|+password]
$CIC	CIC:SYMDEST|user|password|[|TPNAME]
$IMD	IMD:host[+port]|VID|PWP|Adaptor[+arguments]
$COM	COM:server|user|password

For more information on the individual middleware, see the *URB Interfaces Manual*.

The following examples show the corresponding assignments needed in both the client assignment file and the server assignment file.

For a client assignment file:

```
[PATHS]
$MW = TUX:server+13000
;$MW = ENC:||
[SERVICES_EXEC]
; these services are located on server (and transactional)
CURRENT MW:CURRENT
LOAN $MW:LOAN
SAVINGS MW:SAVINGS
[TM]
MANAGER = TUX
;MANAGER = ENC
PATHS = $MW
TIMEOUT= 10
```

For a server assignment file for TUXEDO:

```
[PATHS]
; User name and password are not needed
$DATA = INF:dbms||
[ENTITIES]
*.BANKING DATA:*.*
[SERVICES_EXEC]
; All services are located here (and transactional)
* $SRU:*
[TM]
MANAGER = TUX
PATHS = $DATA, $SRU
```

TXRETURNPHASE=1

For a server assignment file for Encina:

```
[PATHS]
; No username and password needed here (only the DATABASE name)
$TMDATA ORA:||

[ENTITIES]
*.MODEL    $TMDATA:*.*

[TM]
MANAGER = ENC
PATHS = $TMDATA
; For ORACLE + ENCINA
LOGON$TMDATA = Oracle_XA+Acc=P/scott/tiger+SesTm=0
```

; For INFORMIX + ENCINA
;LOGON$TMDATA = db

Chapter 5 EcoTOOLS configuration

This chapter provides information on how to configure your UNIFACE servers and Web applications to work with EcoTOOLS.

5.1 Introduction

EcoTOOLS provides extensive performance and event management capabilities for your network applications.

A System Management Server (SMS) process on each remote node gathers information on the UNIFACE processes running on its node. As information is collected by the SMS process, it is forwarded, via a TCP/IP connection, to an EcoTOOLS agent on the same remote node. Depending on how it is configured, the EcoTOOLS agent may contact the SMS once, and leave the connection open, or it may connect and disconnect at preset intervals.

Each EcoTOOLS agent forwards its data to the EcoTOOLS Management Station. The Management Station uses the information received from its agents to detect problems, and to initiate corrective actions to ensure system performance and continuous availability.

The following UNIFACE products can be monitored using EcoTOOLS:

- PolyServer
- Application Server
- Message Daemon
- Name Server
- Web applications

For more information on EcoTOOLS, see your EcoTOOLS documentation.

5.2 Using EcoTOOLS monitoring

This section describes how to start and stop the SMS process, as well as the assignment setting used to configure EcoTOOLS monitoring support.

Installing EcoTOOLS monitoring support

The executables used to start and stop the SMS process are automatically installed when you install UNIFACE. The executables are located in the **`\bin`** (or **`/bin`** for UNIX) directory of the UNIFACE installation directory.

For information on installing EcoTOOLS, see your EcoTOOLS installation documentation.

Starting the SMS process

In the Microsoft Windows NT environment, use the following command to start the SMS process:

`{start} smservr {SMS_SRV_PORT=`*PortNo*`} {LOGNAME=`*LogName*`}`
`{OPENEXTEND|OPENCLEAR}`

In a UNIX environment, use the following command to start the SMS process:

`smservr {SMS_SRV_PORT=`*PortNo*`} {LOGNAME=`*LogName*`}`
`{OPENEXTEND|OPENCLEAR} {&}`

Where:

- The optional `start` command runs the SMS process in the background. If this command is not specified, the SMS process runs in the foreground. This command is only available for Microsoft Windows NT. (For UNIX, the optional shell (`&`) command runs the SMS process in the background.)

  If the SMS process is not run in the background, you must stop the SMS process (with the `smshut` command) in another window.
- *PortNo* specifies the port number to be used; default is port 15015.
- *LogName* specifies the name and (optionally) the path of the log file created by the SMS process. If no log file is specified, the file **`SMSERV.LOG`** is created in the current directory.
- OPENCLEAR specifies that the existing log file should be overwritten, and OPENEXTEND specifies that the new information should be appended to the existing log file. The default is OPENCLEAR.

Stopping the SMS process

For both the Microsoft Windows NT and UNIX environments, use the following command to stop the SMS process:

`smshut {SMS_SRV_PORT=`*PortNo*`}`

Where:

- *PortNo* specfies the port to be used; default is port 15015.

Assignment settings

The following assignment settings control the behavior of the SMS process, and must be included in the assignment file of the UNIFACE product being monitored:

- `$SMS_REQUIRED` – starts EcoTOOLS monitoring.
- `$SMS_INTERVAL` – specifies the SMS recording interval, that is, the time (in minutes) that data is accumulated locally before being forwarded to the EcoTOOLS agent. The default is five minutes.
- `$SMS_SRV_ADDR` – specifies the full (host and port) address to which the SMS process should listen. The default is *Host*+15015, where *Host* is the name of the current system.

For more information on assignment settings, see the *UNIFACE Reference Manual*.

5.3 Limitations

The following limitations currently apply when using EcoTOOLS monitoring:

- The EcoTOOLS agent relies upon the TCP/IP network protocol to communicate with the SMS process. Hence, the remote node must support TCP/IP.
- Monitoring of the Component Server is not currently supported.
- EcoTOOLS monitoring can only operate in report mode. That is, it can gather information about UNIFACE processes running on a node, but cannot take automatic corrective action.
- The SMS process must run on the same system as the UNIFACE product being monitored.
- If a UNIFACE process, that is, a server or a Web application, is not using the default SMS port address (15015), the UNIFACE and SMS processes must be using the same port address.

Chapter 6 Report Writer Interface (RWI) configuration

To make it easier for you to install and configure UNIFACE on Microsoft Windows and Microsoft Windows NT, the contents of this chapter now appear in the *Microsoft Windows Installation Guide*.

Chapter 7 Microsoft Windows configuration

To make it easier for you to install and configure UNIFACE on Microsoft Windows and Microsoft Windows NT, the contents of this chapter now appear in the *Microsoft Windows Installation Guide*.

Chapter 8 UNIX and MPE/iX configuration

To make it easier for you to install and configure UNIFACE products on UNIX platforms, the contents of this chapter now appear in the *UNIX and MPE/iX Installation Guide*.

Chapter 9 OpenVMS configuration

To make it easier for you to install and configure UNIFACE products on OpenVMS platforms, the contents of this chapter now appear in the *OpenVMS Installation Guide*.

Chapter 10 Macintosh configuration

To make it easier for you to install and configure UNIFACE products on Macintosh platforms, the contents of this chapter now appear in the *Macintosh Installation Guide*.

Chapter 11 OS/2 configuration

To make it easier for you to install and configure UNIFACE products on OS/2, the contents of this chapter now appear in the *OS/2 Installation Guide*.

Appendix A UNIFACE Server Monitor

The UNIFACE Server Monitor, available in UNIX and Microsoft Windows NT environments, allows you to monitor and control the status of the UNIFACE daemons and processes that are active in your network. These include:

- The UNIFACE Name Server on each node
- The UNIFACE Message Daemon on each node
- The UNIFACE Application Servers for asynchronous component execution on each node

(These daemons and processes are discussed in section 4.2 *Assignments for a distributed environment* and section 4.3 *Using the UNIFACE Name Server*.)

The Monitor is invoked from the command line as follows:

`pdmon` {`/cmd=`*Command* {*Options*}} `/dnp=`*Protocol*

Where:

- The commands available with `/cmd=`, as well as the *Options* available for each `/cmd=`*Command*, are shown in table A-1. The *Options* can be provided in any order.
- If `/cmd=` is not present, `/cmd=info` is the default.
- For `/cmd=` and all *Options*, the leading slash (/) can be replaced by a hyphen (-). For example, `-cmd=info` is also valid.
- *Protocol* is the network protocol to use to access the requested daemon or process on the selected host. For example, `TCP:`.

Table A-1 Commands and options available with /cmd= switch. ***part 1 of 2***

Command	Other options available	Description	Default
/cmd=cinf			
Ask the UNIFACE Message Daemon on the specified host to display conversation information.	{`/hst=`*Hostname*}	*Hostname* is the name of the host where the Message Daemon is running.	Local machine
	{`/lsn=`*Port*}	*Port* is the port where the Message Daemon is attached.	13013
`/cmd=info`			
Ask the UNIFACE Message Daemon on the specified host to display information about the asynchronous Application Servers.	{`/hst=`*Hostname*}	*Hostname* is the name of the host where the Message Daemon is running.	Local machine
	{`/lsn=`*Port*}	*Port* is the port where the Message Daemon is attached.	13013
`/cmd=tran`			
Ask the UNIFACE Name Server daemon on the specified host to display the result of assigning the component *ComponentName*. This allows you to test the assignment file translations that are currently established.	`/asx=`*ComponentName*	*ComponentName* is the name of the component to be assigned.	—
	{`/sec=`*ComponentType*}	*ComponentType* is `SVC` or `RPT`, indicating that *ComponentName* should be translated using the assignment file section [SERVICES_EXEC] or [REPORTS_EXEC], respectively.	`SVC`
	{`/hst=`*Hostname*}	*Hostname* is the name of the host where the Name Server is running.	Local machine
	{`/lsn=`*Port*}	*Port* is the port where the Name Server is attached.	14014
`/cmd=init`			
Ask the UNIFACE Name Server daemon on the specified host to reload its assignments from the global and local assignment file. (See section 4.3.2 *Assignments on the Name Server side*.)	{`/hst=`*Hostname*}	*Hostname* is the name of the host where the Name Server is running.	Local machine
	{`/lsn=`*Port*}	*Port* is the port where the Name Server is attached.	14014

Table A-1 Commands and options available with /cmd= switch. ***part 2 of 2***

Command	Other options available	Description	Default
`/cmd=shut`			
Ask the UNIFACE Message Daemon on the specified host to shut down the specified asynchronous Application Server.	{`/hst=`*Hostname*}	See `/cmd=info`.	Local machine
	{`/lsn=`*Port*}	See `/cmd=info`.	13013
	{`/ust=`*UnifaceServerType*}	The symbol used to indicate the server type when the path to the Application Server was opened, usually ASV. (See section 4.2.1 *Assignments on the client side*.)	ASV
	{`/usr=`*UserName*}	*UserName* is the name of the user who started the UNIFACE Server to be shut down. Both the `/ust` and `/usr` options must be specified for a server to be shut down. Otherwise, no action is taken.	Current user
`/cmd=stop`			
Stop the UNIFACE Message Daemon or the UNIFACE Name Server daemon on the specified host. (A daemon that has been stopped must be restarted manually.)	{`/dmn=`*Daemon*}	*Daemon* is `UMD` or `UNS`.	`UMD`
	{`/hst=`*Hostname*}	*Hostname* is the name of the host where the Message Daemon or the Name Server is running.	Local machine
	{`/lsn=`*Port*}	*Port* is the port where the Message Daemon or the Name Server is attached.	For `UMD`, 13013 For `UNS`, 14014
`/cmd=help`			
Display a summary of available Monitor commands.	—	—	—

Examples

- To get ‘help’ information for the Monitor, use the following command:

```
pdmon /cmd=help
```

- To get information about the UNIFACE Servers on the current machine (default port), use either of the following commands:

```
pdmon
pdmon /cmd=info
```

- To test the UNIFACE Name Server translation mechanism, where the Name Server is using port 14015 (rather than the default port 14014), use the following command:

```
pdmon /cmd=tran /lsn=14015 /asx=myownreport /sec=rpt
```

- To ask the UNIFACE Name Server to reload its assignment files, where the Name Server is using port 14015 (rather than the default port 14014), use the following command:

```
pdmon /cmd=init /lsn=14015
```

- To stop the UNIFACE Name Server, running on node phoenix, while logged onto node roc, use the following command:

```
pdmon /cmd=stop /dmn=UNS /hst=phoenix
```

- To shut down a UNIFACE Application Server for user ‘test’ on the local machine, use the following command:

```
pdmon /cmd=shut /ust=ASV /usr=test
```

Appendix B Microsoft Windows initialization files

This appendix presents a complete list of the sections, and their settings, that can be specified in the **`usys.ini`** file. For information on the changes to this file between UNIFACE V6.1, V7.1, and V7.2, refer to *Migration to UNIFACE V7.2*.

B.1 [ACCELERATORS]

This section allows you to define the logical to physical mapping used for menu acclelerators. Using this, you can map a logical function, such as File–>Open, to a keyboard sequence such as Control+F7. Each entry has the syntax:

logical_name=*keystroke_combination*

where:
logical_name is the name used in the Menu Definition form. Logical names that begin with `udbg_` designate Debugger menu options.

keystroke_combination is the accelerator displayed next to the menu item when the menu item is displayed. Key modifiers (Shift, Alt and Control) can be connected with the plus (+), minus (-) or space characters. Keys and modifiers are case-insensitive.

For example:

```
fileopen=Ctrl+O
udbg_start=Alt-G
```

Note: Some keys are reserved and should not be redefined because they have a standard meaning when using Microsoft Windows.

B.2 [APPLICATION]

This section allows you to define the icon and logo used by your application. The following settings are available:

ICON

Specifies the name of the **`.ico`** file that is to be used as the application icon when the application (or session panel) is minimized. This setting does not influence the icon displayed by the Programs menu. The Programs menu takes the icon directly from the **`.exe`** file and is not influenced by this setting. To change the icon in the Programs menu, click Icon in the Properties dialog of the Programs menu.

LOGO16

Specifies the name of an image to be used as the application start-up logo on 16-color displays. To display glyphs as the 16-color logo, use the ^*GlyphName* syntax. To display bitmaps as the 16-color logo, use the @*FileName* syntax. If no 16-color logo is specified, the logo supplied with UNIFACE is used.

LOGO2

Specifies the name of an image file to be used as the application start-up logo on monochrome displays. To display glyphs or bitmaps as the monochrome logo, use the ^*GlyphName* or @*FileName* syntax. If no monochrome logo is specified, the logo supplied with UNIFACE is used.

LOGO256

Specifies the name of an image file to be used as the application start-up logo on 256-color displays. To display glyphs as the 256-color logo, use the ^*GlyphName* or @*FileName* syntax. If no 256-color logo is specified, UNIFACE uses the 16-color logo instead.

LOGOTIME

Specifies the time, in seconds, that the logo is displayed. Allowed values are `0` through `60`. The default is `5`.

B.3 [BACKGROUND] and [FOREGROUND]

These sections determine the mapping of UNIFACE screen attributes to screen colors. Each section defines four palettes of eight colors, each specified as an RGB (red, green, or blue) triple.

The color of an object consists of a foreground and a background color, each of which is taken from its own section. The combined state of the bright and blink video attributes determines the palette that is used. The color number (`0` through `63`) determines which color of each palette is used.

When specifying colors, foreground colors will be mapped to the nearest color that can always be displayed as a 'solid' color by the display device being used. This limits your choice to the 16 or 20 colors in the Microsoft Windows system palette.

These sections can also be changed via the Colors option in the Setup menu. The default color settings supplied with UNIFACE are shown in table B-1:

Table B-1 Default UNIFACE color settings.

Color	Foreground color	Background color
System default	0	0
Blue	8	1
Green	16	2
Cyan	24	3
Red	32	4
Purple	40	5
Yellow	48	6
White	-	7
Black	56	-

B.4 [DB3]

This section is used by the dBase III driver. For more information on supported settings for dBase III, refer to the *dBase III Driver Guide*.

B.5 [DNT]

This section specifies control information used by the PathWorks (DNT) driver. It has the following setting:

MAXREC

Controls the maximum size of records used by the DECnet (DNT) driver. Allowed values are `50` through `8192`. The default is `4000`.

B.6 [GFP]

This section specifies control settings used by the component editor. It has the following settings:

NEWFORM

Specifies the size and location used by the component editor when a new component is created. The format is as follows:

`NEWFORM` = *x_pos*, *y_pos*, *width*, *height*

where *x_pos* and *y_pos* are window coordinates for the top left corner of the component, and *width* and *height* specify the dimensions of the component.

WIDGETS

Specifies the logical widgets that will appear on the component editor's palette, rather than on the palette's drop-down list of available widgets.

B.7 [GRAPHIC FILTERS]

This section specifies the graphic filter DLLs that allow UNIFACE to support particular image file formats. Each entry in this section has the following format:

f={path}filter.flt,ext,dialog

where:

- *f* is the image subtype. For example, a Q defines image type IQ.
- *path* optionally specifies the location of the graphics filter.
- *filter* specifies the name of the graphics filter DLL.
- *flt* specifies the extension of the graphics filter DLL, usually **`.flt`**.
- *ext* specifies the default file extension used for the images handled by the graphics filter.
- *dialog* specifies whether the control dialog of the filter (if it has one) is used. Allowed values are `on` and `off`.

FILTER

Specifies the names of any external filters that you want to use. It has the following format:

`FILTER{`*A-Z*`}=`*filtername*

where:

- *A-Z* is the external filter you have defined with an image type of IA through IZ.
- *filtername* specifies the name of the external filter.

Note: If you intend to write your own image filters for use with Microsoft Windows 95 or Microsoft Windows NT, ensure that they are ALDUS Windows compliant. Export the functions defined in your filters using the `_declspec (dllexport)` *convention.*

B.8 [HELP]

This section specifies the files to be used when online help is requested within the Development Environment or within displayed messages. It has the following settings:

DEFAULT

Specifies the file to be used when online help is requested within UNIFACE. If no help file is specified, no help is available.

MESSAGES

Specifies the file to be used when online help is requested on displayed messages. If no help file is specified, no help is available.

B.9 [HISTORY]

This section records the most recent command lines used to start UNIFACE. A maximum of 10 command lines can be recorded. For example:

```
1=/bat /tst
2=/asn=test.asn
```

UNIFACE automatically prevents the creation of duplicate entries for the same command line history. The Microsoft Windows 95 and Microsoft Windows NT versions of UNIFACE store this information in the Registry.

B.10 [INSTALL]

This section contains information about the current installation. This section is only used by the Microsoft Windows 3.*x* version of UNIFACE. The Microsoft Windows 95 and Microsoft Windows NT versions of UNIFACE store this information in the Registry. It has the following settings:

PROJECT

Specifies the directory in which UNIFACE will place all your compiled components and applications. This directory is also the default directory in which UNIFACE applications are started, and where files are created by UNIFACE utilities whenever a complete path is not specified. The default is **`C:\USYS\PROJECT`**.

ROOT

Specifies the root UNIFACE installation directory. The default is **`C:\USYS`**.

VERSION

Specifies the current version of UNIFACE (7.2).

B.11 [LAYOUT]

This section is used to specify information that is used for form layouts.

HISTORY

Specifies whether a form containing a split bar is opened in the same location that it had when it was last closed. If set to `on`, the position of the form and split bar are saved to the Windows Registry. If set to `off`, the form state is stored in memory only and is lost when you close UNIFACE. The default value is `on`.

B.12 [OCX]

This section is used to specify information about the environment in which OCXs will run. It has the following setting:

OCXTRACE

Specifies the OCX messages that are directed to the Transcript Window.

Allowed values are summed by adding the desired message codes shown in table B-2:

Table B-2 Codes for selecting OCX messages in the Transcript Window.

Code	Description
0	No messages logged.
1	Log error messages (default)
2	Log warning messages
4	Log information messages

For example, setting OCXTRACE to `6` would result in all warning and information messages being logged. The default is `1`, only error messages are logged.

USERMODE

Specifies the setting of the ambient UserMode property for all OCXs in a UNIFACE application. Allowed values are `DESIGN` and `RUN`.

If set to `DESIGN`, the UserMode for all OCXs is set `off`. If set to `RUN`, the UserMode is set `on`. If this setting is not specified, the setting for UserMode depends on how the application is started. If the form component is run within the Development Environment using File–>Test, UserMode is set `off`; in all other cases, UserMode is set `on`.

B.13 [PATHS]

This sections specifies the path names of various software components. It can contain the following settings:

HELPDIR

Specifies the name of the directory where the UNIFACE online documentation is located. This setting is only relevant if you have installed the UNIFACE online documentation. The default directory is the **`\help`** subdirectory of the installation directory.

IMAGES

Specifies the name of the directory where image files are stored. You can specify more than one directory by using a comma (,) separated list. The default is the working directory. UNIFACE always looks in the current directory first. If the image file cannot be found, UNIFACE searches the directories specified by this setting. If the image is still not found, an error is displayed. If you use an explicit path to specify an image, this setting is ignored.

USYS

Specifies the name of the UNIFACE system directory. The default is the **USYS** subdirectory in the parent directory of the **BIN** subdirectory. If this can not be resolved, the default is the current directory. The Software Enable Key (SEK) files and **usys.asn** must always be in the **USYS** subdirectory. Many other files are assumed to be in the **USYS** subdirectory as well, but can be moved to a different location by adding an entry to **usys.asn**. These files include the UOBJ and USYSANA tables, the UNIFACE run-time forms, and all UNIFACE forms.

WORKDIR

Specifies the name of the current directory during the UNIFACE session. This will be the directory where forms and database tables are created, unless overridden with an assignment. Microsoft Windows allows you to specify the working directory in the Program's Properties. This is overridden by a WORKDIR specification. The default is the current directory.

B.14 [PRINTER] and [SCREEN]

These sections specify the fonts available for printing and displaying, respectively. There are separate sections for screen and printer fonts, because a font that is suitable for the screen may print very slowly if it does not match exactly with a built-in printer font. If you want true 'what you see is what you get' (WYSIWYG), you should make both sections identical.

When the three character styles (bold, italic and underline) are all enabled in the [UPI] section, only one font can be specified. When a style is disabled (it is recommended that you disable italic), two fonts can be specified. When two styles are disabled, four fonts can be specified, and when all styles are disabled, all eight fonts can be used.

It is recommended that you select only fixed-pitch fonts. Selecting very large fonts will not give attractive results, as all characters are positioned (and clipped) in a fixed-size character grid.

If you are using an enhanced printer device translation table, you can also specify logical to physical font mappings in the [PRINTER] section. If you want to use a particular physical font for displaying information, and another font for printing, define the same logical font in each section, but use a different physical font in each mapping.

By default, UNIFACE uses the same logical fonts for displaying and printing. However, these logical fonts are mapped to different physical fonts in the [PRINTER] and [SCREEN] sections of the **`usys.ini`** file. Each entry has the following format:

logical_font=*font*,*script*,*pitch*,*style*

For example:

```
font0=Courier New,Western,9,regular
```

Logical printer mapping

The [PRINTER] section also allows you to define the logical to physical printer mapping used by UNIFACE. Each entry takes the following form:

logical_printer=*physical_printer*

For example:

```
myprinter=\PRINTERS\HPLASERJ 4Si
landsc=\PRINTERS\HPLASERJ 4Si
```

Within the Registry, mappings are held in the [PRT_*logical_printer*] subkey.

You can define more than one logical name to be mapped to the same physical printer. However, you cannot define multiple logical printers with the same name. It is recommended that you do not edit the contents of this section. Instead, use the Printer item on the Setup menu to define your logical to physical printer mappings.

B.15 [SINGLE_INSTANCE]

This section is used to control whether applications can have more than one instance running. It has the following syntax:

application=*window_title*

When a new instance of a UNIFACE application is started, UNIFACE first checks whether that application is listed under the [SINGLE_INSTANCE] section. If so, it looks for a window which starts or ends with the specified title. The title string is not case sensitive. If such a window is found, the focus changes to the existing application, and a second instance is not started.

For example, to prevent a second instance of the UNIFACE Development Environment from being started, enter the following definition:

```
[SINGLE_INSTANCE]
IDF=UNIFACE Seven
```

B.16 [STATE]

This section records the state of certain UPI objects when the user exits an application. All settings in this section are automatically updated by UNIFACE whenever it terminates.

GFPPALETTE

Determines the visibility of the Tool Palette within the Development Environment. Allowed values are 0 (`off`) or 1 (`on`). The default is 1.

GFPSTATUS

Determines the visibility of the Status Box within the Development Environment. Allowed values are `0` (`off`) or `1` (`on`). The default is `1`.

PANEL

Determines the visibility of the Session Panel and the toolbar. Allowed values are `on` or `off`. The visibility can be set `on` or `off` at run time using the Panel... command on the application panel, or with GOLD+X. The default is `on`.

PANELMIN

Records whether the panel was last minimized or not. Allowed values are `on` or `off`. The default is `off`.

PANELPOS

Holds the position of the floating session panel as a horizontal position, vertical position pair. This setting is only used if the panel position in the start-up shell form is set to default. The default is `0,0`.

PANELSIZE

Holds the size of the floating Session Panel as a width, height pair. The default is `0,0`.

WINDOWMAX

Determines the initial state of the application window (using the entire screen or not). Allowed values are `on` or `off`. The default is `off`.

WINDOWPOS

Holds the position of the application window as a horizontal position, vertical position pair. The default is the default Microsoft Windows position.

WINDOWSIZE

Determines the initial size of the application window. The application window can be resized at run time. The values should be supplied as a pair of width, height values. The default is the default Microsoft Windows window size.

B.17 [TCP]

This section is used by the TCP/IP drivers. It can contain the following settings:

ETC

Specifies the location of the Services file. This setting is used by the FTP TCP/IP driver.

MAXREC

Controls the maximum size of records used by all TCP/IP drivers. Allowed values are `256` through `8192`. The default is `4096`.

SOCKETS

Specifies the maximum number of open channels. The default is `2`. This setting is used by the SUN NFS TCP/IP driver.

TIMEOUT

Specifies the maximum time (in seconds) before a time-out error is generated. The default is `0`, which means an infinite amount of time. This setting is used by the FTP TCP/IP driver.

TCPSTRIPPEDDOMAINRETRY

Specifies whether UNIFACE should first attempt to connect to the target host using any specified domain information and, if this fails, remove the domain information from the target host name and try again to connect.

For more information on the use of this setting, see the *Microsoft Windows Installation Guide*.

UPSVINACTIVETCP

Specifies the maximum wait time, in minutes, on a TCP/IP network path for a server. If the client has been inactive for the specified wait time, the server is automatically terminated.

If the UPSVINACTIVETCP setting is not specified, the server continues running regardless of how long the client is active. The maximum wait time that can be specified is appropriately 18 hours. It is recommended that you specify a wait period suitable for your working environment (for example, eight hours).

For more information on the use of this setting, see the *Microsoft Windows Installation Guide*.

UTCP_RB

Sets the maximum TCP receive buffer size in kilobytes.

UTCP_SB

Sets the maximum TCP send buffer size in kilobytes.

UTCP_NAGLE

Causes the default behavior of the TCP_NODELAY option to be disabled. This is a boolean option and can be set to any non-zero value.

B.18 [TOOLBAR]

This section controls the availability of the various toolbar controls. It can contain the following settings:

CHARSETS

Determines the availability of a selection button on the far left-hand side of the toolbar for each of the character sets 0 through 7. Allowed values are eight comma-separated 0/1 values; default is `1,1,1,1,1,0,0,1`.

CSETACCELERATOR

Controls whether the character set can be set using the shortcuts Alt+0 through Alt+7. Allowed values are `0`, `1`, or `2`. When set to `0`, the shortcuts are not available. When set to `1`, the shortcuts are only available when the toolbar is visible. When set to `2`, the shortcuts are always available. The default is `2`.

FONTLIST

When set `on`, a combo box listing the available fonts is displayed on the toolbar. This is only useful when more than one font is available. The default is `on`.

MENUTOGGLE

When set `on`, the Menu Toggle button will be displayed on the far right-hand side of the toolbar. This button will be disabled (dimmed) when only one menu is available. The default is `off`.

MDICONTROLS

When set `on`, the MDI combo box, tile, and cascade buttons are shown on the toolbar. The default is `off`.

STYLEBUTTONS

When set `on`, the Bold, Italic and Underline buttons are shown on the toolbar. One or more of these buttons may be disabled (dimmed), depending on the setting of bold, italic, and underline in the [UPI] section. The default is `on`.

VIEWTOGGLE

Allowed values are `on` or `off`. When `on`, a View button is displayed on the toolbar. The default is `on`.

B.19 [UDDE]

This section controls the use of Dynamic Data Exchange (DDE) support from within a UNIFACE application. It has the following settings:

DDE

Controls whether DDE support is available within a UNIFACE application. Allowed values are `on` and `off`. The default is `off`.

TIMEOUT

Controls the length of time (in seconds) that can elapse before a DDE transaction will generate a time-out error. The default is `1`.

B.20 [UNIFACE_DLLS]

This section specifies the DLLs used by UNIFACE. It has the following settings:

DEMANDLOAD

Specifies the DLLs that UNIFACE should search when it needs a particular 3GL function. Separate DLL names with commas or semicolons. If you do not specify a full path name for a DLL, UNIFACE loads it from the directory specified in the PATH setting. This setting is always taken from the default **`.ini`** file.

You can load a maximum of 32 DLLs as DEMANDLOAD with a maximum string size of 512 characters.

NLS

Specifies the National Language Support (NLS) used by UNIFACE. You must have the appropriate version of Microsoft Windows installed for a particular NLS module to function. If you select an NLS module that does not exist, or if the NLS module detects that the version of Microsoft Windows for which it was designed is not running, UNIFACE automatically reverts to the build-in USA version. This setting can be taken from either the local or default **`.ini`** file.

PATH

Specifies the directory where UNIFACE DLLs are located. The default is the working directory. If no directory is specified here, the current directory, the DOS path, the Microsoft Windows and the SYSTEM directories are searched in that order. This increases application load time (if the DLLs are found at all). This setting is always taken from the default **`.ini`** file.

PRELOAD

Specifies the DLLs that UNIFACE should load at start-up, regardless of whether it needs any of the functions in the DLL. Separate DLL names with commas or semicolons. If you do not specify a full path name for a DLL, UNIFACE loads it from the directory specified in the PATH setting. This setting is always taken from the default **`.ini`** file.

You can load a maximum of 32 DLLs as PRELOAD with a maximum string size of 512 characters.

B.21 [UPI]

This section determines the appearance or behavior of many Universal Presentation Interface (UPI) objects under Microsoft Windows. The following settings are available:

AUTOIMECLOSE

Controls whether the Input Method Editor (IME) is disabled when the focus changes to a field that only allows input in Font 0. Setting AUTOIMECLOSE `on` disables the IME. The default is `on`.

When a read-only field receives focus, the status of the IME is unchanged. If the field is defined as Special String, the IME is enabled; otherwise, it is disabled.

AUTOIMEGOLD

Controls whether the IME is disabled when the GOLD key is used. Once the complete GOLD key sequence has been entered, the IME is returned to its previous state. Setting AUTOIMEGOLD `on` disables the IME when the GOLD key is used. The default is `on`.

AUTOIMEOPEN

Controls whether the IME is automatically enabled when the focus moves to a field that allows double-byte characters (such as kanji).

When a read-only field receives focus, the status of the IME is unchanged. If the field is defined as Special String, the IME is enabled; otherwise, it is disabled. The default is `off`.

AUTOMAXIMIZE

Controls whether forms are automatically maximized when they do not fit in the application window. This setting also controls whether a component that was previously maximized is maximized again when it is run again. Allowed values are `on` or `off`. If set `off`, components are not automatically maximized, unless the Quick Zoom function is used. The default is `on`.

BOLD

The settings Bold, Italic, and Underline determine if the respective character attributes are available or are translated to a different font instead. Allowed values are `on` or `off`. The default is `on`.

BUTTONSPACING

Determines the number of pixels between adjacent buttons on a panel. The value can range from `0` through `10`. The default is `2`.

BUTTONSTYLE

Specifies the appearance of buttons and borders on the toolbar and panels. Allowed values are shown in table B-3:

Table B-3 BUTTONSTYLE allowed values.

Value	Description
0	Normal Microsoft Windows 3.*x* appearance. For Microsoft Windows 3.*x*, this is the only supported value.
1	Normal Microsoft Windows 95 appearance. This is the default for Microsoft Windows 95 and Microsoft Windows NT.
2	The Microsoft Windows 98 (flat toolbar) appearance.

BUTTONTYPE

The type of button on session and component panels. Allowed values are `Text` or `Icon`. The default is `Text`.

CELLHEIGHT

Specifies the cell height (as a percentage of the cell height of Font 0) used to paint widgets. The default is `100`, that is, use the same cell height as Font 0. Allowed values are `100` and `200`.

COLORINVERSE

Controls the color value used as the default for inverse color objects. The default is `7`.

COLORNORMAL

Controls the color value used as the default for normal color objects. The default is `56`.

DEBUGLINES

The number of lines remembered by the list box in the Proc debugger. Allowed values are `5` through `100`. The default is `100`.

FMTFORMPANELS

Allowed values are `on` or `off`. This setting is like FMTSESSIONPANEL, but applies to the component-level panels. The default is `off`.

FMTSESSIONPANEL

Determines whether the formatting information that is embedded in the Session Panel is honored or ignored. For portability, the default is `off` (that is, ignore). If you have configured text buttons, this setting is ignored and formatting is always applied. The default is `off`.

FOCUSFLASH

Allowed values are `on` or `off`. When `on`, a flashing rectangle indicates where the keyboard focus is, when it is on a button or image. The default is `on`.

FRAMEALIGN

Determines whether a frame, such as an area, line, or named area frame, is painted on the left edge of a cell (so that it can be aligned with widget borders), or centered. Allowed values are `Center` or `Left`. The default is `Center`.

FRAMETYPE

The type of graphic used to draw frames on components. Allowed values are `Normal`, `Etched`, or `Shaded`. Note that the `Etched` and `Shaded` types do not look attractive on a white background. The `Shaded` type also has some inherent problems with intersecting frames. The default is `Etched`.

ITALIC

Allowed values are `on` or `off`. The default is `off`.

KEYPADENTERACCESS

Specifies whether the keypad Enter key is routed through the keyboard translation table for command buttons. Allowed values are `on` (keypad Enter key activates the default command button), or `off` (key is routed through keyboard translation table. The default is `off`.

LINESPACE

Allows you to specify additional spacing between text lines in the background. Recommended values are between `0` and `3`. The default is `3`.

MINRESOURCE

The percentage of system resources that UNIFACE should try to leave available to other applications. When this percentage is reached, UNIFACE will release some resources before it allocates new ones; notice that components are minimized automatically. The default value is `70` (percent), but any value between `10` and `90` is acceptable. The resources referred to are in the data segment of the USER module.

MSGLINES

The number of messages remembered by the message line. Allowed values are `5` through `100`. The default is `100`.

PAINTCACHE

Allowed values are `on` or `off`. When `on`, the paint cache accelerates screen updates at the expense of increased memory consumption. The default is `off`.

PANELTOOLTIP

Specifies whether tool tip text (when available) appears for icon buttons in panels. Allowed values are `on` or `off`. The default is `on`.

PANELTYPE

Determines the location of the Session Panel. Allowed values are `Top`, `Left`, `Bottom`, `Right`, `Floating`, and `None`. The default is `Floating`.

POPUPDELAY

Determines the time delay between pressing the MENU mouse button and the appearance of the pop-up menu. If this setting is not specified at all, the default used is the double-click time set in the Microsoft Windows Control Panel. Allowed values are the time (in milliseconds) of the delay.

PROFILE

Determines the prefix character used when displaying profile characters in widgets (but not unifields). Allowed values are ANSI character codes. The default is `165` (Japanese), `41380` (Chinese and Korean), or `183` for all other character sets.

SCROLLBARS

Determines the appearance of the entity scroll bars on components. Allowed values are `Narrow`, `Standard`, or `Aligned`. The default is `Standard`.

Narrow scroll bars fit in the width of a single character cell, and can therefore be used on existing components. Standard scroll bars have the Microsoft Windows standard width of a scroll bar. While more attractive, they may be too wide for existing (crowded) components. Aligned scroll bars are standard Microsoft Windows scroll bars, rounded to the nearest whole cell width.

SOUND

Enables or disables all beeps and buzzes generated by UNIFACE. Allowed values are `on` or `off`. The default is `on`.

TEXTFRAME

Editable text fields can optionally have a border. This can make components dramatically more attractive. The available types of border are shown in table B-4:

Table B-4 Allowed values of TEXTFRAME. ***part 1 of 2***

Setting	Effect
None	No border
Plain	Narrow black line

Table B-4 Allowed values of TEXTFRAME. ***part 2 of 2***

Setting	Effect
Etched	Makes the field appear sunken
Bevelled	Makes the field appear raised
Shaded	A more native shading effect

`Etched` and `Bevelled` work best when the background of the component is not white or dark gray, because these are the colors used to draw the frame. On a white background, `Shaded` looks quite attractive. Note that fields that are not editable do not have a border. The default is `Etched`.

TRANSLINES

The number of lines remembered by the Transcript Window before it starts losing lines at the top. If this setting is not specified at all, a default of 100 is used. The default setting in the **`.ini`** file of 16,000 should be adequate for most applications. The allowed values are `100` through `16380`. The default is `100`.

UNDERLINE

Allowed values are `on` or `off`. The default is `on`.

WINDOWONUNIFIELD

Enables or disables overlaying a Unifield with a window. The default value is `off` for both 16-bit Windows and 32-bit Windows.

Note: For 16-bit Windows applications, setting the value to `on` *may result in insufficient memory. Therefore, this switch should only be set to* `on` *when using a testing application, such as QARun, that needs the window handle.*

B.22 [USER]

This optional section contains the user name and optionally the password of the user who runs the current workstation. The following settings are available:

NAME

Specifies the user name with which a user will log on. The default is an empty string.

PASSWORD

Specifies the password with which a user will log on.

Both of these settings are available in Proc through the `$user` and `$password` functions. If accessed via Proc, the password is not encrypted. The default is an empty string.

B.23 [USERDLLS]

This section has the following settings:

DEMANDLOAD

Specifies the user-defined DLLs that UNIFACE should search when it needs a particular 3GL function. Separate DLL names with commas or semicolons. If you do not specify a full path name for a DLL, UNIFACE loads it from the directory specified in the PATH setting.

You can load a maximum of 32 DLLs as DEMANDLOAD with a maximum string size of 512 characters.

PATH

Specifies the directory where user-defined DLLs are located. The default is the working directory. If no directory is specified here, the current directory, the DOS path, the Microsoft Windows and System directories are searched in that order. This increases application load time (if the DLLs are found at all).

PRELOAD

Specifies the user-defined DLLs that UNIFACE should load at start-up, regardless of whether it needs any of the functions in the DLL. Separate DLL names with commas or semicolons. Use this setting to list DLLs that can do their job without explicitly being called by UNIFACE, or when you want to avoid DLL loading delays halfway through your application.

You can load a maximum of 32 DLLs as PRELOAD with a maximum string size of 512 characters.

B.24 [WIDGETS]

This section is used to make the mapping between logical and physical widgets. Entries in this section have the following format:

logical_name=*physical_name*{(*property*$_1$=*value*$_1$; *property*$_2$=*value*$_2$;...)}

Appendix C X resources

This appendix presents a complete list of the X resources used by the OSF/Motif version of UNIFACE to customize the appearance and behavior of applications.

C.1 Introduction

The X resources used by UNIFACE are specified in the **`xdefault.txt`** file (**`xdef_jpn.txt`** for kanji), located in the UNIFACE installation directory. All UNIFACE applications have the class 'Uniface'. For example, to define that all UNIFACE applications should have a white foreground and a black background, the following resources are set:

```
Uniface*background: black
Uniface*foreground: white
```

This not only affects the UNIFACE Development Environment, but all applications generated with it. The rules of precedence used by X dictate that actual application names take precedence over classes. For example, the following settings make the Development Environment appear as black on white, and all other UNIFACE applications as white on black:

```
Uniface*background:  black
Uniface*foreground:  white
idf*background:      white
idf*foreground:      black
```

Table C-1 shows a summary of the resources used by the OSF/Motif version of UNIFACE:

Table C-1 Summary of UNIFACE X resources. ***part 1 of 3***

X resource/class	Description
Font resources	
`xfont`*n*	Normal font rendering for `font`*n*.
`xfont`*n*`b`	Bold font rendering for `font`*n*.
`xfont`*n*`i`	Italic font rendering for `font`*n*.
`xfont`*n*`bi`	Combined bold and italic rendering for `font`*n*.
`ufileselectionbox`	Specifies the attributes for fonts used within selection boxes generated by the `filebox` instruction.
`udebugbox`	Specifies the attributes of fonts used in debug dialog boxes.
`upanelbutton`	Specifies the attributes of fonts used within control panel buttons.
`umsgdialog`	Specifies the attributes of fonts used within dialog boxes generated by the `askmess` statement.
`uradiodialog`	Specifies the attributes of fonts used to display radio button text within dialog boxes generated by the `askmess` statement.
`statuswindow`	Specifies the attributes of fonts used in the floating Status Window.
Panel position resources	
`uappanelon`	Specifies whether panels are displayed.
`uappanelposition`	Specifies the position of panels.
`uformpanelon`	Specifies whether form component panels are displayed.
`uformpanelposition`	Specifies the position of form components.
Color resources	
`foreground_`*n*	Application foreground color definition for color_*n*.
`foreground_`*n*h	Application highlighted foreground color definition for color_*n*.
`foreground_`*n*b	Application blinking foreground color definition for color_*n*.
`background_`*n*	Background color definition for color_*n*.
`background_`*n*h	Application highlighted background color definition for color_*n*.

Table C-1 Summary of UNIFACE X resources. ***part 2 of 3***

X resource/class	Description
`background_`*n*`h`	Application blinking background color definition for color_*n*.
`colorscheme`	Specifies color inheritance of a widget.
Menu resources	
`accelerator`	Specifies the keystroke sequences for accelerator keys.
`acceleratorText`	Specifies the text that appears on menu items for accelerator keys.
`menubar`	Specifies the attributes of the attributes of the (horizontal) menu bar.
`menuoption`	Specifies the attributes of options in the menu bar.
`menuoptionseparator`	Specifies the attributes of the pull-down menu’s separator.
`pulldownmenu`	Specifies the attributes of all pull-down menu items.
`pulldownmenuoption`	Specifies the attributes of individual items on a pull-down menu.
IME resources	
`uimopen`	Specifies whether the IME is automatically enabled for a field that supports double-byte input.
`uimgold`	Specifies whether the IME is disabled when the GOLD key is used.
`uimclose`	Specifies whether the IME is disabled when focus moves to a field that only allows input in font0.
`uimstyle`	Specifies the level of support for the native IME.
Miscellaneous	
`image_cache`	Specifies whether images are retained in a cache or removed from memory after a form component is closed.
`imagefile_path`	Specifies directory to search for image files.
`title.fontlist`	Specifies window title font (independent of window manager).
`Mwm*title*fontList`	Specifies window title (for the OSF/Motif window manager).

Table C-1 Summary of UNIFACE X resources. ***part 3 of 3***

X resource/class	Description
`ToggleVision`	Specifies the visual marker used within the check box widget.
`MessageDialog`	Specifies the attributes of message boxes generated by the `askmess` statement.
`profile`	Specifies the prefix character used for displaying profile characters in widgets.
`ufileshell`	Specifies the attributes of file selection boxes generated by the `filebox` statement.
`uvextra`	Specifies the additional space left between lines on a form component.
`use_icons_in_control_panel`	Specifies whether text or icons appear on command buttons in a control panel.
`old_key_behavior`	Specifies whether, for compatibility reasons, former keyboard translation table handling is used.

C.2 Font resources

A font under X governs the way that a particular character is rendered. This includes the basic shape of the character (Avant Garde, Times, Helvetica, and so on), the size (10, 12, 14 point, and so on), the orientation (oblique, italic, and so on) and many other attributes. X uses a 14-part name to identify a particular font; fortunately some of the parts can be replaced with asterisks (*).

UNIFACE can use a large number of different fonts in an application. For example, one font can be used for bold characters, one font for italicized characters. These fonts are specified as X11 resources. UNIFACE supports eight internal fonts, numbered 0 through 7, for western European characters. See the *UNIFACE Reference Manual* for more information on character sets.

The basic UNIFACE fonts are specified as follows:

`Uniface*xfont`*n*`:`*X font name*

where *n* is the UNIFACE font number (0 through 7).

For example, to set Font 0 to Courier, 14 point and to use ISO 8859-1 characters, the following definition would be used:

```
Uniface*xfont0: -adobe-courier-medium-r-normal--14-*-*-*-*-*-iso8859-1
```

UNIFACE supports different character renderings, for italic, bold, and combined bold and italic. These are indicated by the suffixes ‘b’, ‘i’, and ‘bi’, respectively. For example, the following settings define all possible Font 0 and Font 1 renderings:

```
Uniface*xfont0:   -adobe-courier-medium-r-normal--14-*-*-*-*-*-iso8859-1
Uniface*xfont0b:  -adobe-courier-bold-r-normal--14-*-*-*-*-*-iso8859-1
Uniface*xfont0i:  -adobe-courier-medium-o-normal--14-*-*-*-*-*-iso8859-1
Uniface*xfont0bi: -adobe-courier-bold-o-normal--14-*-*-*-*-*-iso8859-1
Uniface*xfont1:   -dec-terminal-medium-r-normal--14-*-*-*-*-*-dec-dectech
Uniface*xfont1b:  -dec-terminal-bold-r-normal--14-*-*-*-*-*-dec-dectech
Uniface*xfont1i:  -dec-terminal-medium-o-normal--14-*-*-*-*-*-dec-dectech
Uniface*xfont1bi: -dec-terminal-bold-o-normal--14-*-*-*-*-*-dec-dectech
```

Use the `xlsfonts` command to list all the available fonts on your system.

C.2.1 ufileselectionbox

The class name for the fonts used within selection boxes generated by the `filebox` Proc instruction is `ufileselectionbox`. You can specify the following resources:

- `labelFontList`: The font used for labels within selection boxes.
- `buttonFontList`: The font used in buttons within selection boxes.
- `textFontList`: The font used for all text (other than label and button text) within selection boxes.

For example, to set the font for all labels displayed within selection boxes generated by the `filebox` Proc instruction, you would use the following definition:

```
Uniface*ufileselectionbox*labelFontList:
-adobe-helvetica-medium-r-normal-14-140-75-75-p-77-iso8859-1
```

C.2.2 udebugbox

The class name for the fonts used within the debug dialog box is `udebugbox`. This is the dialog box activated using the `debug` Proc statement. You can specify the following resources:

- `labelFontList`: The font used for labels within the debug dialog box.
- `buttonFontList`: The font used for buttons within the debug dialog box.
- `fontList`: The font used for title text within the debug dialog box.
- `textFontList`: The font used for all text (other than label, button and window title text) within the debug dialog box.

For example, to set the font used for all labels appearing in the debug dialog box, you would use the following definition:

```
Uniface*udebugbox*labelFontText:
-adobe-helvetica-medium-r-normal-14-140-75-75-p-77-iso8859-1
```

C.2.3 upanelbutton

The class name for all the fonts used within pop-up panel buttons is `upanelbutton`. It has one resource, `fontList`, which specifies the font used to display text within the buttons.

For example:

```
Uniface*upanelbutton*fontList:
-adobe-helvetica-medium-r-normal-14-140-75-75-p-77-iso8859-1
```

C.2.4 umsgdialog

The class name for the fonts used with dialog boxes generated by the `askmess` Proc statement is `umsgdialog`. You can specify the following resources:

- `labelFontText`: The font used for label text within the dialog box.
- `buttonFontText`: The font used to display text within the command buttons in the dialog box.
- `textFontList`: The font used to display all text (other than label and command button text) within the dialog box.

For example, to set the font used for label text within the dialog box:

```
Uniface*umsgdialog*labelFontList:
-adobe-helvetica-medium-r-normal-14-140-75-75-p-77-iso8859-1
```

C.2.5 uradiodialog

The class name for the fonts used to display radio button text within dialog boxes generated by the `askmess` Proc statement is `uradiodialog`. It has one resource, `fontList`, that specifies the font used to display radio button text.

For example:

```
Uniface*uradiodialog*fontlist:
-adobe-helvetica-medium-r-normal-14-140-75-75-p-77-iso8859-1
```

C.2.6 statuswindow

The class name for the fonts used by the floating Status Window of the component editor is `statuswindow`. The Status Window shows the current status of a selected object.

The font used for the heading labels (shown on the left) is specified by the `titlelabel.fontList` resource.

The font used for the values (shown on the right) is specified by the `valuelabel.fontList` resource.

For example, to set the heading and value fonts for the Status Window, the following definitions would be used:

```
! define title font for Status Window
Uniface*statuswindow*titlelabel.fontList:-b&h-typewriter-medium-r-normal-sans-12
-*-*-*-*-*-iso8859-1

!define values font for Status Window
Uniface*statuswindow*valuelabel.fontList:-b&h-typewriter-medium-r-normal-sans-10
-*-*-*-*-*-iso8859-1
```

C.2.7 Logical font mappings

The mapping of logical widget font names to physical font names is specified by using X resources. The logical font name is given on the left-hand side of the assignment, the physical name on the right-hand side. For example, to define the logical font `efont`, the following X resource setting would be made:

```
Uniface*efont: -b&h-lucidatypewriter-medium-r-normal-sans-10-*-*-*-*-*-iso8859-1
```

The available logical fonts are specified by the `fonts` X resource. For example:

```
Uniface*fonts:   efont, lfont, label, gfp
```

C.3 Panel position resources

The following UNIFACE application resources are available to control the position of the application pop-up panel on the application screen or a form component:

C.3.1 uappanelon

If a panel has been defined for an application in the Start-up Shell Properties form, the `uappanelon` resource can be used to control the display of the panel. Allowed values are True and False. For example:

```
Uniface*uappanelon: False
```

C.3.2 uappanelposition

If the position of a panel has not been defined in the Define Start-up Shell Properties form, the `uappanelposition` resource allows you to set its position at run time. Allowed values are Top, Bottom, Left, and Right. For example:

```
Uniface*uappanelposition: Top
```

C.3.3 uformpanelon

If a panel has been defined for a form component, the `uformpanelon` resource can be used to define whether the panel is to be displayed. Allowed values are True and False. For example:

```
Uniface*uformpanelon: True
```

C.3.4 uformpanelposition

If the position of a form component has not been defined in the Define Form Component Properties form, the `uformpanelposition` resource allows you to set its position at run time. Allowed values are Top, Bottom, Left, and Right. For example:

```
Uniface*uformpanelposition: Top
```

C.4 Color resources

UNIFACE allows you to set the color values for the following types of data display:

- Normal
- Blinking
- Highlighted
- Blinking and highlighted

Note: Most graphical displays do not support blinking characters.

There are four color palettes that can be used with a UNIFACE application. All the colors in all the palettes can be defined via the setting of various X resources.

Normal color definitions

The normal colors are set by the resources `foreground_0` through `foreground_7`, and `background_0` through `background_7`. For example:

```
Uniface*foreground_0:black
Uniface*background_0:white
```

Highlight color definitions

The colors used for fields containing highlighted information are set by the resources `foreground_0h` through `foreground_7h`, and `background_0h` through `background_7h`. For example:

```
Uniface*foreground_0h:red
Uniface*background_0h:white
```

Blinking color definitions

The colors used for fields containing blinking information are set by the resources `foreground_0b` through `foreground_7b`, and `background_0b` through `background_7b`. For example:

```
Uniface*foreground_0b:pink
Uniface*background_0b:white
```

Most graphical displays do not actually support characters blinking.

Highlighted blinking color definitions

The colors used for fields containing highlighted blinking information are set by the resources `foreground_0hb` through `foreground_7hb`, and `background_0hb` through `background_7hb`. For example:

```
Uniface*foreground_0hb:green
Uniface*background_0hb:white
```

C.4.1 colorscheme

If you do not explicitly define the background color of a widget, it is taken from its parent. You can change this behavior with the use of the `colorscheme` resource. If this resource is not set, or is set to `InheritFromParent`, widgets continue to take their background color from their parents. If you set the colorscheme resource to `UserSpecified`, the color is set using the background resource.

For example, the following definition would set the default background color for all edit boxes that have explicit color setting to white:

```
Uniface*ueditbox.colorscheme: UserSpecified
Uniface*ueditbox.background: white
```

C.4.2 CDE color compliance

If you intend to use Common Desktop Environment (CDE) to control the display of UNIFACE applications, it is recommended that you set all color combinations in your UNIFACE applications to the system default. This is the color combination that is used when you set the foreground color and the background color to the same color, for example, if you specify that background color is 0, and foreground color is 0.

You can use the StyleManager utility within CDE to specify control and font settings. For more information on how to control the appearance of applications under CDE, refer to your CDE documentation.

C.5 Menu resources

This section describes the resources that can be used to customize the menus used by the OSF/Motif version of UNIFACE.

C.5.1 accelerator and acceleratorText

The OSF/Motif version of UNIFACE uses X resources to define the logical to physical mapping for menu bar accelerators. For each menu bar item, both the accelerator keys and the accelerator text can be specified. Each menu item uses the accelerator defined for it in the Define Window Properties form. This is combined with the instance name for menu items.

To specify the accelerator keys and accelerator text for the menu items defined as, for example, File–>Open, the following definitions would be used:

```
Uniface*pulldownmenu.fileopen.accelerator:      Ctrl<Key>O
Uniface*pulldownmenu.fileopen.acceleratorText:  Ctrl + O
```

The `accelerator` setting specifies the keystroke sequence (specified in the standard Motif way). The `acceleratorText` setting specifies the text that will appear on the menu item showing the accelerator keys.

C.5.2 menubar

The `menubar` resource specifies the attributes of the (horizontal) menu bar. For example, to specify that the menu bar items should appear with red text on a white background, the following definitions would be made:

```
Uniface*menubar*foreground:                     red
Uniface*menubar*background:                     white
```

C.5.3 menuoption

The `menuoption` resource specifies the attributes of options in the menu bar. You can specify attributes for all instances of menu bar items, or specific named instances of them.

For example, to specify that the Edit option should appear with blue text on a yellow background, the following definitions would be made:

```
Uniface*edit.foreground:            blue
Uniface*edit.background:            yellow
```

To set the backgrounds for all the other menu bar items to black, and the foregrounds to white, the following definitions would be made:

```
Uniface*menuoption*foreground:          white
Uniface*menuoption*background:          black
```

C.5.4 menuoptionseparator

The `menuoptionseparator` resource specifies the attributes of the pull-down menu's separator. For example, to specify that the pull-down menu's separator should be painted three pixels wide, the following definitions would be made:

```
Uniface*menuoptionseperator.shadowThickness:    3
```

The separators used by UNIFACE are derived from the `XmSeparator` widget. Consequently, all the resources supported by this widget can be used with this resource.

C.5.5 pulldownmenu

The `pulldownmenu` resource specfies the attributes of all pull-down menu items. You can use this to set resources for all pull-down menus, or for specific items using the resources described in the following sections.

For example, to make all pull-down menus appear with a white background, you would use the following definition:

```
Uniface*pulldownmenu*background:      white
```

C.5.6 pulldownmenuoption

The `pulldownmenuoption` resource specifies the attributes of individual items on a pull-down menu. For example, to specify that all Close items should appear with red text, the following definition would be made:

```
Uniface*pulldownmenu.close.foreground:red
```

The example shows how the foreground color of an instance of the `pulldownmenuoption` class (in this case Close) is set.

C.6 IME resources

The following sections describe the resources used to control the operation of the Input Method Editor (IME).

C.6.1 uimopen

The `uimopen` resource controls whether the host Input Method Editor (IME) is automatically enabled when focus on a form component is given to a special string (that is, one that supports double-byte characters). The default is False.

For example:

```
Uniface*uimopen: True
```

C.6.2 uimgold

The `uimgold` resource controls whether the host IME is disabled when the GOLD key is used. When the complete GOLD key sequence has been entered, the IME is returned to its previous state. Setting `uimgold` to True disables the IME when the GOLD key is used. The default is True.

For example:

```
Uniface*uimgold: True
```

C.6.3 uimclose

The `uimclose` resource controls whether the host IME is disabled when focus changes to a field that only allows input in font0. Setting `uimclose` to True disables the IME when the focus changes to a field that only allows input in Font 0.

For example:

```
Uniface*uimclose: False
```

C.6.4 uimstyle

The `uimstyle` resource controls the style support for the native IME. The styles are defined by the X library and are as follows:

- XIMPreeditArea (1)
  The client provides geometry management of an area in which the native IME can perform over-the-spot editing.
- XIMPreeditCallbacks (2)
  The client provides pre-edit callback procedures so that the native IME can cooperate with the application to perform on-the-spot pre-editing.
- XIMPreeditPosition (3)
  The client provides the location of the insertion cursor so that the native IME can perform over-the-spot pre-editing.

Table C-2 Default uimstyle setting for different platforms.

Platform	IME	Setting
HPUX10	XJIM	3
HPUX10	ATOK8	7
HPUX10	VJE_gama	7
HPUX10	ECNBridge	2
IBM 4.1.4	aixims	0
Solaris 2.5	htt	1
AlphaOSF	dxjim	1

C.7 Miscellaneous resources

The following sections describe miscellaneous resources that can be set for the OSF/Motif version of UNIFACE.

C.7.1 MessageDialog

The `MessageDialog` resource specifies the attributes of all message boxes displayed as a result of the `askmess` Proc statement. For example, to make all message boxes appear with a red background, the following setting would be used:

```
! make all message boxes appear with a red background
Uniface*MessageDialog.background:     red
```

The message boxes used by UNIFACE are derived from the `XmMessageBox` OSF/Motif widget. Consequently, all the resources supported by this widget can be used with this resource.

C.7.2 profile

The `profile` resource specifies the prefix character used when displaying profile characters in widgets (but not unifields). For example, to use the % character (ASCII character 37), the following setting would be used:

```
! Use the % character to denote a profile
Uniface*profile:     37
```

C.7.3 ufileshell

The `ufileshell` resource specifies the attributes of all file selection boxes displayed as a result of the `filebox` Proc statement. For example, to make the file selection box appear with a blue background, the following setting would be used:

```
! make all message boxes appear with a blue background
Uniface*ufileshell.background:     blue
```

The file selection boxes used by UNIFACE are derived from the XmFileSelectionBox OSF/Motif widget. Consequently, all the resources supported by this widget can be used with this resource.

C.7.4 uvextra

The uvextra resource sets the additional space, in pixels, that is left clear between lines on a form component. The effect of this is to increase the height of the form component. For example:

```
! Add two pixels to the character cell height
Uniface*uvextra:    2
```

C.7.5 image_cache

The image_cache setting specifies whether images are retained in a cache after the form components that used them have been closed. If set to False, an image is dropped from memory after the form component that used it is closed. This setting is useful when running applications in low memory situations. The default is True. For example:

```
Uniface*image_cache:    False
```

C.7.6 imagefile_path

The image_path resource specifies the directory where UNIFACE looks for its image files. If you do not specify a setting, the $usys setting is used. You can specify multiple paths to search by separating each path with a semicolon (;) character. For example:

```
! Search Tom's and Fred's home directories for images
uniface*imagefile_path:/home/user/tom;/home/user/fred/
```

C.7.7 Mwm*title*fontlist

The Mwm*title*fontlist resource specifies the font to be used for the window title in the Window Manager.

The `Uniface*title*fontlist` X resource specifies the font to be used for the window title, independent of the window manager. The font specified in this resource should be the same as the font specified for the OSF/Motif windows manager `Mwm*title*fontlist` resource, if the OSF/Motif windows manager is used.

```
! Change the font in all UNIFACE applications in the user environment:
Mwm*Uniface*title*fontlist:-adobe-courier-bold-i-normal--0-0-0-0-m-0-iso8859-1

! Change the font in the application called mission_critical_uappl:
Mwm*mission_critical_uappl*title*fontList:adobe-courier-bold-i-normal--0-0-0-0-m
-0-iso8859-1
```

Restart your OSF/Motif window manager (mwm) for your font change to take effect. For any other window manager, replace the application class `Mwm` by the class resource of your host window manager.

The `Mwm*title*fontlist` resource specifies the font to be used for the window title in the window manager. For example:

```
! Render the title on the title area
Mwm*Uniface*title*fontList:-adobe-helvetica-bold-r-normal--14--*-*iso8859-1
```

C.7.8 ToggleVisual

The `ToggleVisual` resource specifies the visual marker used within the check box widget.

If the check box's field value is True, the `NoToggleVisual` resource can have the following values:

- `NoToggleVisual` - No check or mark (default).
- `ToggleVisualMark` - A mark bitmap is displayed in the check box.
- `ToggleVisualCross` - A cross bitmap is displayed in the check box.

For example:

```
uniface*ucheckbox.ToggleVisual:ToggleVisualMark
```

C.7.9 use_icons_in_control_panel

The `use_icons_in_control_panel` resource controls whether text, rather than icons, appears on a command button in a control panel. When set to False, all command buttons in control panels appear with text, rather than icons. The default is True.

For example:

```
Uniface*use_icons_in_control_panel: False
```

C.7.10 old_key_behavior

The `old_key_behavior` resource controls whether the keyboard translation table uses new or old functionality. This new functionality allows key combinations, such as Control+A and Control+Alt+C, to be sent to the keyboard translation table. The `old_key_behavior` resource exists for compatibility reasons. When set to False, the new functionality is used. The default is True.

For example:

```
Uniface*old_key_behavior: False
```

C.8 Printing support

Full printing support is provided for UNIFACE on OSF/Motif. Font and color mappings are specified in the **`psdef.txt`** file located in the **`USYS`** directory of the installation directory.

The file is divided into sections that specify the font mappings, color mappings, and paper size to be used during printing. The logical font and color identifiers are the same as those used in the **`xdefault.txt`** file. The red, green, and blue color values are floating values from 0.00 through 1.00.

The **`psdef.txt`** file has the following default contents:

```
[FONTS]
xfont0:       Courier                 9
xfont0b:      Courier-Bold            9
xfont0i:      Courier-Oblique         9
xfont0bi:     Courier-BoldOblique     9
efont:        Helvetica               9
efontb:       Helvetica-Bold          9
efonti:       Helvetica-Oblique       9
efontbi:      Helvetica-BoldOblique   9
lfont:        Helvetica               9
lfontb:       Helvetica-Bold          9
lfonti:       Helvetica-Oblique       9
lfontbi:      Helvetica-BoldOblique   9
label:        Helvetica               12
```

```
buttonfont:   Helvetica              12

[FOREGROUND]
color_0:      0.00 0.00 0.00   black
color_1:      0.00 0.00 1.00   blue
color_2:      0.00 1.00 0.00   green
color_3:      0.00 1.00 1.00   cyan
color_4:      1.00 0.00 0.00   red
color_5:      1.00 0.00 1.00   magenta
color_6:      1.00 1.00 0.00   yellow
color_7:      0.00 0.00 0.00   black

color_0h:     0.66 0.66 0.66   dark grey
color_1h:     0.00 0.00 0.55   dark blue
color_2h:     0.00 0.39 0.00   dark green
color_3h:     0.33 0.34 0.18   dark green olive
color_4h:     0.42 0.22 0.22   indian red
color_5h:     0.55 0.12 0.55   dark orchid
color_6h:     0.68 1.00 0.18   green yellow
color_7h:     0.66 0.66 0.66   dark grey

color_0b:     1.00 1.00 1.00   white
color_1b:     0.49 0.49 0.49   grey
color_2b:     0.00 1.00 0.00   green
color_3b:     0.00 1.00 1.00   cyan
color_4b:     1.00 0.00 0.00   red
color_5b:     1.00 0.00 1.00   magenta
color_6b:     1.00 1.00 0.00   yellow
color_7b:     1.00 1.00 1.00   white

color_0hb:    0.66 0.66 0.66   light grey
color_1hb:    0.69 0.89 1.00   light blue
color_2hb:    0.45 0.87 0.47   pale green
color_3hb:    0.91 0.59 0.48   salmon
color_4hb:    1.00 0.53 0.00   orange
color_5hb:    1.00 0.71 0.77   pink
color_6hb:    0.20 0.85 0.22   yellow green
color_7hb:    0.74 0.56 0.56   rosy brown

[BACKGROUND]
color_0:      1.00 1.00 1.00   white
color_1:      0.00 0.00 1.00   blue
color_2:      0.00 1.00 0.00   green
color_3:      0.00 1.00 1.00   cyan
color_4:      1.00 0.00 0.00   red
color_5:      1.00 0.00 1.00   magenta
color_6:      1.00 1.00 0.00   yellow
color_7:      1.00 1.00 1.00   white

color_0h:     0.49 0.49 0.49   grey
color_1h:     0.00 0.00 1.00   blue
```

```
color_2h:     0.00 1.00 0.00   green
color_3h:     0.00 1.00 1.00   cyan
color_4h:     0.42 0.22 0.22   indian red
color_5h:     1.00 1.00 1.00   red
color_6h:     1.00 1.00 0.00   yellow
color_7h:     1.00 1.00 1.00   white

color_0b:     0.00 0.00 1.00   blue
color_1b:     0.00 0.00 1.00   blue
color_2b:     0.00 0.00 1.00   blue
color_3b:     0.00 0.00 1.00   blue
color_4b:     0.00 0.00 1.00   blue
color_5b:     0.00 0.00 1.00   blue
color_6b:     0.00 0.00 1.00   blue
color_7b:     0.00 0.00 0.00   blue

color_0hb:    0.00 1.00 1.00   cyan
color_1hb:    0.00 1.00 1.00   cyan
color_2hb:    0.00 1.00 1.00   cyan
color_3hb:    0.00 1.00 1.00   cyan
color_4hb:    0.00 1.00 1.00   cyan
color_5hb:    0.00 1.00 1.00   cyan
color_6hb:    0.00 1.00 1.00   cyan
color_7hb:    0.00 1.00 1.00   cyan

[PAPERS]
A4:    210.0  297.0
```

For more information on printing support, refer to section 6.7 *Enhanced printing device translation tables* in the *UNIFACE Reference Manual*.

Appendix D Compatibility codes

Table D-1 presents a list of the three-letter compatibility codes used to indicate a supported platform in UNIFACE 7.2. For a complete list of supported platforms, you should consult your UNIFACE representative.

Table D-1 Compatibility codes. ***part 1 of 2***

CCODE	Description
A71	DIGITAL Alpha OpenVMS 7.1
AN2	Microsoft Windows NT 4.0 Alpha
AS1	AS/400e, OS/400 V4R1,V4R2,V4R3
DU1	DIGITAL UNIX 4.0
DG3	Data General AViiON Intel, DG/UX 4.20 (ACO)
IB4	IBM OS/2 4.0
HP8	HP9000, HP-UX 10.20/11
MAC	Apple Macintosh 8 (Power Mac only)
MPE	HP3000, MPE/ix 5.5
MS1	Microsoft Windows 95
MSW	Microsoft Windows 3.11
MVS	MVS (OS/390)
NC9	NCR 3000/4000/5000 UNIX MP-RAS 3.02
NT4	Microsoft Windows NT 4.0 Intel
RS4	RS/6000, AIX 4.2.1/4.3
RU1	SNI, RM Series, Reliant Unix 5.43/5.44
SC6	SCO OpenServer 3.2 v5.0.x
SG5	Silicon Graphics, IRIX 6.5 (32 bits)

Table D-1 Compatibility codes. ***part 2 of 2***

CCODE	Description
SO5	Sun SPARC Solaris 2.5.1/2.6
SQ7	Sequent Dynix PTX 4.4
UW3	UnixWare 7
V71	DIGITAL VAX OpenVMS 7.1

Appendix E Security driver

UNIFACE provides a security driver that enables developers to provide their own 3GL implementation to encode and decode user names and passwords when connecting to server on the synchronous path. Currently, only the TCP driver supports the use of the security driver, which is available for the following platforms:

- Microsoft Windows 95/NT
- Alpha NT
- Microsoft Windows 3.11
- UNIX
- OpenVMS

By default, the logon information message contains the username and password for the remote login and is not encoded. You can define your own 3GL to encode and decode this message.

This document describes how to create and install your own security driver.

E.1 Implementing a custom security driver

To implement a customized security driver:

1. Obtain the security driver sources. See section E.2 *Obtaining the security driver sources*.
2. Create your own security driver by modifying the sources to provide encryption and decryption. See section E.3 *Creating a custom security driver*.
3. Install and link your security driver. See section E.3 *Creating a custom security driver*.

For information on the security driver structure and the interface, see section E.4 *Security driver structure* and section E.4.1 *Security driver entry point and functionality*.

The following restrictions apply to the security driver:

- It only encrypts and decrypts the username and password. It is expected that more functionality will be added in the future.
- It only works when using a TCP/IP network connection.
- It is only implemented on Microsoft Windows, UNIX (including MPE/iX), and OpenVMS platforms.
- It is implemented for synchronous network access, so PolyServer and the synchronous Application Server support the driver. The asynchronous Application Server scrambles passwords by default.
- No encryption/decryption algorithms are provide. You must write your own code, or provide a hook into existing encryption software.
- The interface is based on the C language. You can only code the security driver in C or C++. However, from within the C code you can call code written in other languages, as long as you are aware of the calling conventions.

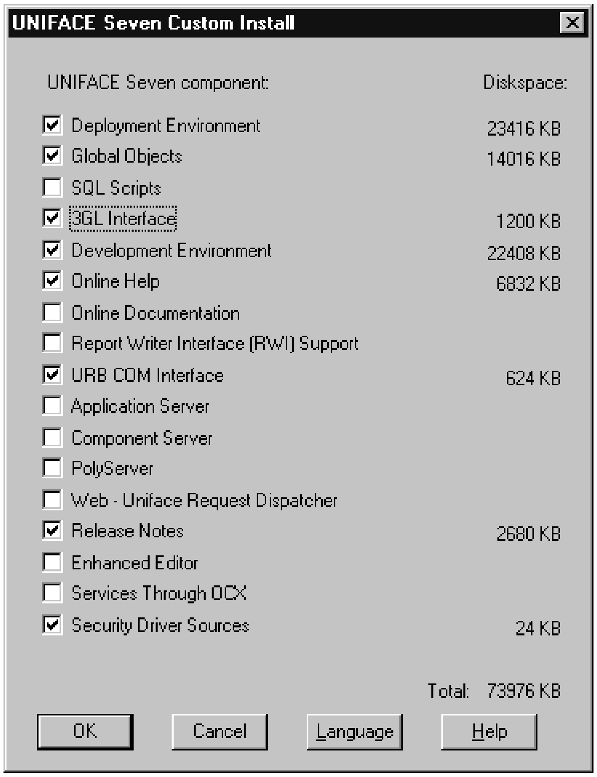
E.2 Obtaining the security driver sources

The default driver is always installed on supported platforms. The source files for the driver are automatically provided on UNIX and OpenVMS. On Microsoft Windows, the source files are provided only if you have used the Custom Installation dialog box during installation to select 3GL Interface and Security Driver Sources.

E.2.1 Obtaining sources on Microsoft Windows platforms

To obtain the security driver on Windows platforms:

1. Start the installation program, as described in the Microsoft Windows Installation Guide.
2. In the first dialog box, click Custom to display the Custom Install dialog box.
3. Select Security Driver Sources and 3GL Interface, as shown in figure 11-1.



UNIFACE Seven Custom Install

UNIFACE Seven component:	Diskspace:
✓ Deployment Environment	23416 KB
✓ Global Objects	14016 KB
☐ SQL Scripts	
✓ 3GL Interface	1200 KB
✓ Development Environment	22408 KB
✓ Online Help	6832 KB
☐ Online Documentation	
☐ Report Writer Interface (RWI) Support	
✓ URB COM Interface	624 KB
☐ Application Server	
☐ Component Server	
☐ PolyServer	
☐ Web - Uniface Request Dispatcher	
✓ Release Notes	2680 KB
☐ Enhanced Editor	
☐ Services Through OCX	
✓ Security Driver Sources	24 KB
Total:	73976 KB

Figure 11-1 Custom Install dialog box.

4. Proceed with the installation, as described in the UNIFACE Microsoft Windows Installation Guide.

Security driver files

After installation, the following source files are located in the **`/bin`** subdirectory of the UNIFACE installation directory:

- Dynamic Link Library (DLL) containing the security driver. The name of the file depends on the platform:
  - Microsoft Windows 95 and NT 4: **`zsecintc.dll`**
  - Windows 3.11: **`zsecint.dll`**
  - Alpha NT: **`zsecintx.dll`**

- **`zsecint.c`**, an example source file.
- **`usecdef.h`**, the header file for the example source file.

Although you will eventually replace the DLL with your own implementation, save the original file so that you can easily restore the default implementation without reinstalling the whole product.

The DLL is automatically added to the Demand Load line in the [UNIFACE_DLLS] section of the **`.ini`** file.

In addition to the security driver source files, several 3GL files can be used when implementing the security driver. These files are located in the **`\3gl`** subdirectory of the UNIFACE installation directory:

- **`\include\umsw3gl.h`**, a header file included with the 3GL Interface that includes a macro required for the security driver.
- **`\samples\makefile.inc`**, a basic makefile suitable for your Windows version and platform that should be included the makefile for your own security driver.

E.2.2 Obtaining sources on UNIX and OpenVMS platforms

On UNIX and OpenVMS, the security driver is a module in the UNIS library. The security driver source files are automatically provided in the **`/bin`** subdirectory of the UNIFACE installation directory.

To obtain the driver, extract the module ZSECINT and save it. This enables you to restore the default implementation without reinstalling the whole product again.

Security driver files

On OpenVMS platforms, the following sources are provided:

- **`zsecint.obj`**, object module in the library **`libunis.olb`**
- **`zsecint.c`**, an example source file.
- **`usecdef.h`**, the header file for the example source file.

On UNIX platforms, the following sources are provided:

- **`zsecint.o`**, object module in the library **`libunis.a`**
- **`zsecint.c`**, an example source file.
- **`usecdef.h`**, the header file for the example source file.

E.3 Creating a custom security driver

The provided security driver example, **zsecint.c**, contains an example of a driver that encodes the synchronous server request message. To activate this functionality, compile the file with the macro `TEST_ENCODE` to encode connection protocol requests.

Use the security driver structure and entry point function described in the next section to customize the source. When you create your own security driver implementation, you must replace the provided DLL or object files with the new implementation.

The security driver can only be used for applications that are deployed on supported platforms: Microsoft Windows 95, Microsoft Windows 3.11, Microsoft Windows NT 4, Alpha NT, UNIX, and OpenVMS. All paths in the deployed application must be updated to use the features of the security driver.

E.3.1 Implementing on Microsoft Windows platforms

To implement the security driver on Microsoft Windows platforms:

1. Modify the provided source file or create your own. For information on the structure and entry point function, see section E.4 *Security driver structure* and section E.4.1 *Security driver entry point and functionality*.

Note: Remember to save the original ***ZSECINT.DLL*** *so that the default implementation can be easily restored without reinstalling the whole product.*

2. Include the **umsw3gl.h**, which defines the `XEXPORT()` macro that specifies the calling convention.
3. Include **makefile.inc** in your own makefile.
4. Compile the source file into a DLL using a supported C compiler.
5. Edit the Demand Load list in **usys.ini** and **psys.ini** to remove the name of the default security driver and specify your own driver name.

Examples

The following example makefile for Windows NT or Windows 95 builds the source **security.c** into a DLL called **security.dll**:

```
LOC = 'define the location of your driver sources'

# define the location of the 3GL Interface
U3GL = $(INSTALL_DIR)\win4\3gl

# include the supplied 3GL Interface makefile
!include $(U3GL)\samples\makefile.inc

# define the default target
default: $(LOC)\security.dll

# assume the security driver include file is also in $(LOC)
$(LOC)\security.obj:$(LOC)\security.c $(LOC)\usecdef.h
    $(CL) -Du_msw -I$(LOC) -Fo$@ $(LOC)\security.c

$(LOC)\security.dll: $(LOC)\security.obj
    $(LINK) -DLL -OUT:$@ @<<
$(LOC)\security.obj
$(LIBS)
<<NOKEEP
```

The following is a Windows 3.1 makefile example:

```
# assume your sources in the current working directory

# define the location of the 3GL Interface
U3GL = $(INSTALL_DIR)\ms1\3gl

# include the 3GL the supplied 3GL Interface makefile
!include $(U3GL)\samples\makefile.inc

# the target
security.dll:security.obj  security.def
    $(LINK) @<<
security.obj
security.dll
security.map
$(LIBS)
security.def
<<NOKEEP
    $(RC) -t security.dll
```

The following example shows the **security.def** file:

```
LIBRARY       SECURITY
DESCRIPTION'  Security driver for UNIFACE SEVEN'
EXETYPE       WINDOWS
CODE          PRELOAD MOVEABLE
DATA          PRELOAD MOVEABLE SINGLE
HEAPSIZE      1024
```

```
STACKSIZE     0
EXPORTS       WEP @1 RESIDENTNAME
        __UCONNECT
```

E.3.2 Implementing on UNIX and OpenVMS platforms

To implement the security driver:

1. Modify the provided source file or create your own. For information on the structure and entry point function, see section E.4 *Security driver structure* and section E.4.1 *Security driver entry point and functionality*.
2. Compile the source file into an object module using a supported C compiler.

   There are no special compilation requirements. Use your default compiler settings to produce one or more object files that contain your driver implementation. These object files can then be added to the UNIS library.
3. Remove the default driver and save it elsewhere.
4. Add your own security driver. On UNIX, use the command `AR` to replace the dummy objects. On OpenVMS, use the command `LIBRARY` to replace the module.
5. Once your driver objects are in the UNIS library, re-run the installation program and choose option 4, which relinks your applications. You must do this for all UNIFACE and PolyServer installations that participate in the new security scheme.

E.4 Security driver structure

Table E-1 lists the available members of the security driver structure. Not all members are described, as some are reserved for UNIFACE use only.

Table E-1 Security driver members.

Member	Description
`void *DrvArea`	Used to allocate memory and store the pointer; also responsible for freeing any memory that it allocates.
`unsigned char *InBuf`	Points to the start of the input buffer, which contains the information to be encoded or decoded.
`int InBufLen`	Length in bytes of the buffer pointed to by `InBuf`
`int InDataLen`	Length in bytes of the data stored in the buffer pointed to by `InBuf`.
`unsigned char *OutBuf`	Points to the start of the output buffer, which contains the result of encoding or decoding the data in the input buffer `InBuf`.
`int OutBufLen`	Length in bytes of the buffer pointed to by `OutBuf`
`int OutDataLen`	Length in bytes of the data stored in the buffer pointed to by `OutBuf`.
`unsigned char *ErrMsg`	Points to a buffer containing the error message text that describes the supplied error number. This buffer is guaranteed to be at least 256 bytes long.

Table E-1 Security driver members.

Member	Description
`int ErrMsgLen`	Length in bytes of the error message text in the buffer pointed to by `ErrMsg`
`long Error`	Error number returned by the security driver if it encounters an error. It should also return a status from the entry point function indicating the severity of the error. This member is only viewed when the entry point function return status is not successful. It is this number for which text must be supplied in the `ErrMsg` buffer.
`unsigned char Function`	This member describes the action to be taken by the security driver. The following actions are defined: • `USEC_INFO`—Specify this implementation's functionality • `USEC_ENCODE`—Encode the contents of `InBuf` and place the result in `OutBuf` • `USEC_DECODE`—Decode the contents of `InBuf` and place the result in `OutBuf` • `USEC_ERRMSG`—Translate the error number into text For more information on these actions, see section E.4.3 *Function codes*.

E.4.1 Security driver entry point and functionality

WINDOWS platforms

The syntax of the security entry point function for Windows is:

```
#include "usecdef.h"
XEXPORT(long) USECINT(USec *Sec)
```

where:

`XEXPORT()` is a macro provided in **`umsw3gl.h`**, a file provided with 3GL Interface option of the installation. It specifies the calling convention. For more information, see section E.3 *Creating a custom security driver*.

`USECINT` is the entry point function.

`(USec *Sec)` is a pointer to the security driver structure described in section E.4 *Security driver structure*.

UNIX and OpenVMS

The syntax of the security entry point function for UNIX and OpenVMS is:

```
#include "usecdef.h"
long USECINT(USec *Sec)
```

where:

`USECINT` is the entry point function

`(USec *Sec)` is a pointer to the security driver structure described in section E.4 *Security driver structure*.

E.4.2 Return values

The function returns a long value, which must be one of the following:

- `USEC_SUCCESS`—Everything worked.
- `USEC_ERROR`—Something went wrong. The error is returned and the user can decide what to do. The action can be retried or not, depending on the application context.
- `USEC_FATAL`—Something went wrong in such a way that there is no point in trying again. The application returns the error and then exits. The entry point function must examine the Function member and determine what action needs to be taken. The functionality of the driver is expected to be expanded, so function codes that are not recognized should be returned as errors.

E.4.3 Function codes

This section describes the function codes that can be encountered.

USEC_INFO

The caller is requesting information about the functionality of the driver. The following macros are provided:

- `USetSecConnectEnc(USec *Sec)`
  Use this macro to specify that this security driver implementation encodes the synchronous server request message. After using this macro, your security driver will receive encode and decode requests.
- `UZeroSecConnectEnc(USec *Sec)`
  Use this macro to specify that this security driver implementation does not encode the synchronous server request message. The default driver implementation uses this macro.
- `UIsSecConnectEnc(USec *Sec)`
  Use this macro to determine whether the driver encodes the synchronous server request. The initial setting in the structure is that no encoding will take place, so this macro returns false. If `USetSecConnectEnc()` has been called, this macro returns true. The entry point function is not allowed to return an error with this function code.

USEC_ENCODE

The caller is requesting the data specified by `InBuf` and `InDataLen` to be encoded. The encoded data must be placed in `OutBuf` and its length, which must be less than or equal to `OutBufLen`, should be returned in `OutDataLen`.

For a synchronous server request message, substantial enlargement of data due to encoding is supported. The `OutBuf` is approximately three times the size of the `InBuf`. The complete input data must be encoded and placed in the output buffer. If the output buffer is not large enough to hold the encoded data, modify the `InDataLen` member to indicate how much of the input has been encoded. For a synchronous server request message, this results in an error because the message must be sent in an unsegmented form. (In the future, this same mechanism will be used to segment large buffers that require encoding.)

USEC_DECODE

The caller is requesting the data specified by `InBuf` and `InDataLen` to be decoded. The decoded data must be placed in `OutBuf` and its length, which must be less than or equal to `OutBufLen`, should be returned in `OutDataLen`. If the output buffer is not large enough to hold the decoded data, modify the `InDataLen` member to indicate how much of the input buffer has been decoded.

USEC_ERRMSG

The driver must provide an error message that describes the error identified by the `Error` member of the driver structure. The textual message should not exceed 256 bytes in length and its length should be returned in `ErrMsgLen`.

Although the message format is entirely implementation-specific, the macro USEC_ERR_HEADER is provided. This macro defines the null-terminated string, `"Security Driver Error [%d] : %s"`. It is a format string that can be given to functions like `printf()`. It requires the error number and the error message as arguments.

Index