Rocket Model 204 Horizon:

Intersystem Processing Guide

Version 7 Release 5.0

September 2014
204–75–HZN-01

www.rocketsoftware.com

Notices

Edition

Publication date: September 2014
Book number: 204–75–HZN-01
Product version: Version 7 Release 5.0

Copyright

Trademarks

Examples

This information might contain examples of data and reports. The examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

License agreement

Note: This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

Corporate Information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Rocket Global Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA

Contacting Technical Support

If you have current support and maintenance agreements with Rocket Software and CCA, contact Rocket Software Technical support by email or by telephone:

Email: m204support@rocketsoftware.com

Telephone :

North America	+1.800.755.4222
United Kingdom/Europe	+44 (0) 20 8867 6153

Alternatively, you can access the Rocket Customer Portal and report a problem, download an update, or read answers to FAQs. You will be prompted to log in with the credentials supplied as part of your product maintenance agreement.

To log in to the Rocket Customer Portal, go to:

www.rocketsoftware.com/support

Contents

About this Guide

1 How to Use This Guide

2 Model 204 Intersystem Processing Fundamentals

3 Network Management

4 Security

7 Horizon User Language (SOUL) Sample Programs

8 Horizon Error Processing

A Horizon Conversation States and Statements

B LU 6.2 Verb Set Equivalences

C SNA Communications Server Performance Tuning for Horizon

D Connecting to Non-Model 204 Systems

E Horizon CNOS Connections

Index

About this Guide

Audience

This guide is designed to meet the needs of application programmers, the Model 204 system manager, network administrators, and systems programmers.

A note about User Language and SOUL

Model 204 version 7.5 provides a significantly enhanced, object-oriented, version of User Language called SOUL. All existing User Language programs will continue to work under SOUL, so User Language can be considered to be a subset of SOUL, though the name "User Language" is now deprecated. In this guide, the name "User Language" has been replaced with "SOUL."

Purpose of this guide

The *Rocket Model 204 Horizon: Intersystem Processing Guide* describes the intersystem processing facilities of Model 204, with primary attention to Horizon. Horizon permits Model 204 SOUL application programs running in separate processors to converse as peers across an IBM SNA Communications Server (formerly VTAM)-controlled SNA network. The conversation interface consists of SOUL statements that conform to the LU 6.2 protocol.

Model 204 documentation set

To access the Rocket Model 204 documentation, see the Rocket Documentation Library (http://docs.rocketsoftware.com/), or go directly to the Rocket Model 204 documentation wiki (http://m204wiki.rocketsoftware.com/).

Notation conventions

This guide uses the following standard notation conventions in statement syntax and examples:

Convention	Description
`TABLE`	Uppercase represents a keyword that you must enter exactly as shown.
`TABLE` *`tablename`*	In text, italics are used for variables and for emphasis. In examples, italics denote a variable value that you must supply. In this example, you must supply a value for *tablename*.
`READ [SCREEN]`	Square brackets ([ ]) enclose an optional argument or portion of an argument. In this case, specify READ or READ SCREEN.
`UNIQUE | PRIMARY KEY`	A vertical bar (|) separates alternative options. In this example, specify either UNIQUE or PRIMARY KEY.
`TRUST | NOTRUST`	Underlining indicates the default. In this example, NOTRUST is the default.
`IS {NOT | LIKE}`	Braces ({ }) indicate that one of the enclosed alternatives is required. In this example, you must specify either IS NOT or IS LIKE.
`item ...`	An ellipsis (. . .) indicates that you can repeat the preceding item.
`item ,...`	An ellipsis preceded by a comma indicates that a comma is required to separate repeated items.
`All other symbols`	In syntax, all other symbols (such as parentheses) are literal syntactic elements and must appear as shown.
`nested-key` `::=` *`column_name`*	A double colon followed by an equal sign indicates an equivalence. In this case, *nested-key* is equivalent to *column_name*.

`Enter your account:`
`sales11`
	In examples that include both system-supplied and user-entered text, or system prompts and user commands, boldface indicates what you enter. In this example, the system prompts for an account and the user enters **sales11**.

File > Save As	A right angle bracket (>) identifies the sequence of actions that you perform to select a command from a pulldown menu. In this example, select the Save As command from the File menu.
EDIT	Partial bolding indicates a usable abbreviation, such as E for EDIT in this example.

1

How to Use This Guide

Overview

The *Rocket Model 204 Horizon: Intersystem Processing Guide* describes Horizon, the distributed application facility from Rocket Software. This document provides the information necessary for various types of users to implement and use Horizon. This guide is a handbook for programmers, network administrators, and Model 204 system administrators.

Relationship to Model 204 Documentation Set

The *Rocket Model 204 Horizon: Intersystem Processing Guide* is the main source of information for Horizon. The Model 204 documentation set does not include information specific to Horizon that is not also included in this guide.

However, this guide is not completely a standalone document. It does make references to Model 204 documents for more information about aspects of Model 204 that relate to the use of Horizon. For example, this guide refers to the Rocket Model 204 documentation wiki SOUL/User Language pages for information about creating and using Model 204 application subsystems.

Other locations of Horizon information

In addition to this guide, the Model 204 documentation set discusses aspects of Horizon in several sources, including the following:

- Rocket Model 204 installation information (for IBM z/OS, IBM z/VM, or IBM z/VSE)
- Rocket Model 204 documentation wiki System management pages
- Rocket Model 204 documentation wiki SOUL/User Language pages: http://m204wiki.rocketsoftware.com/index.php/SOUL
- Rocket Model 204 documentation wiki command pages: http://m204wiki.rocketsoftware.com/index.php/Category:Commands
- Rocket Model 204 documentation wiki parameter pages: http://m204wiki.rocketsoftware.com/index.php/List_of_Model_204_parameters

Documentation map

This guide is designed to meet the needs of the following types of individuals involved with the Horizon product:

- Application programmers
- Model 204 system managers
- Network administrators
- Systems programmers

All users should read Chapter 2 to get an overview of Horizon and to become familiar with the terminology used in this guide.

Note: Systems programmers responsible for installing Horizon should refer to the Rocket Model 204 installation information for their operating system.

Application programmer chapters

The following sections are particularly relevant to application programmers developing or modifying distributed Model 204 applications using Horizon:

- Chapter 5: “Horizon Conversation Interface”
- Chapter 8: “Horizon Error Processing”
- Appendix A: “Horizon Conversation States and Statements”
- Appendix B: “LU 6.2 Verb Set Equivalences” (helpful for programmers who have developed applications in other LU 6.2 environments)
- Appendix D: “Connecting to Non-Model 204 Systems”

Network administrator chapters

The following sections are particularly relevant to persons responsible for establishing and maintaining a network supporting Horizon:

- Chapter 3: “Network Management”
- Chapter 4: “Security”
- Appendix D: “Connecting to Non-Model 204 Systems”
- Appendix E: “Horizon CNOS Connections”

Model 204 system manager chapters

The following sections are particularly relevant to Model 204 system administrators (depending on their responsibilities):

- Chapter 3: “Network Management”
- Chapter 4: “Security”
- Chapter 6: “Horizon SOUL Interface”
- Appendix D: “Connecting to Non-Model 204 Systems”

Systems programmer chapters

The following sections are particularly relevant to systems programmers responsible for the IBM SNA Communications Server (formerly VTAM) network:

- “Defining the Network to SNA Communications Server” on page 36
- Appendix C: “SNA Communications Server Performance Tuning for Horizon”
- Appendix D: “Connecting to Non-Model 204 Systems”
- Appendix E: “Horizon CNOS Connections”

2

Model 204 Intersystem Processing Fundamentals

Overview

This chapter introduces Model 204 intersystem processing options and provides a discussion of basic concepts and terminology. A review of this chapter may be in order for both experienced and inexperienced intersystem programming personnel, since the conventions followed by Model 204 may differ from those to which you may be accustomed.

Terms and Concepts

This section introduces some of the basic networking terminology used in this document and illustrates a simple intersystem connection involving Model 204.

Network: a collection of nodes

In this guide, a computer ***network*** is defined as a collection of ***nodes***, connected, at the software level, by a communication interface. Some examples of communication interfaces are:

- The Virtual Telecommunication Access Method (VTAM)
- The Model 204 Cross Region Access Method (CRAM)
- The VM Inter User Communication Vehicle (IUCV)

Strictly speaking, there are different types of nodes on a network: application nodes, terminal nodes, printer nodes, and so on. When the term ***node*** is used in this guide, however, it refers to an application node, that is, the point of connection to the network for some system that supports application programs. Examples of such systems follow:

- Model 204
- CICS
- CMS
- PC DOS or OS/2

Network diagram

Figure 2-1 on page 7 is a diagram of a hypothetical network.

It consists of three nodes and the software communication interface SNA Communications Serverdoc. The network links system M204A with system M204B; the transactions originate from terminals X, Y, and Z.

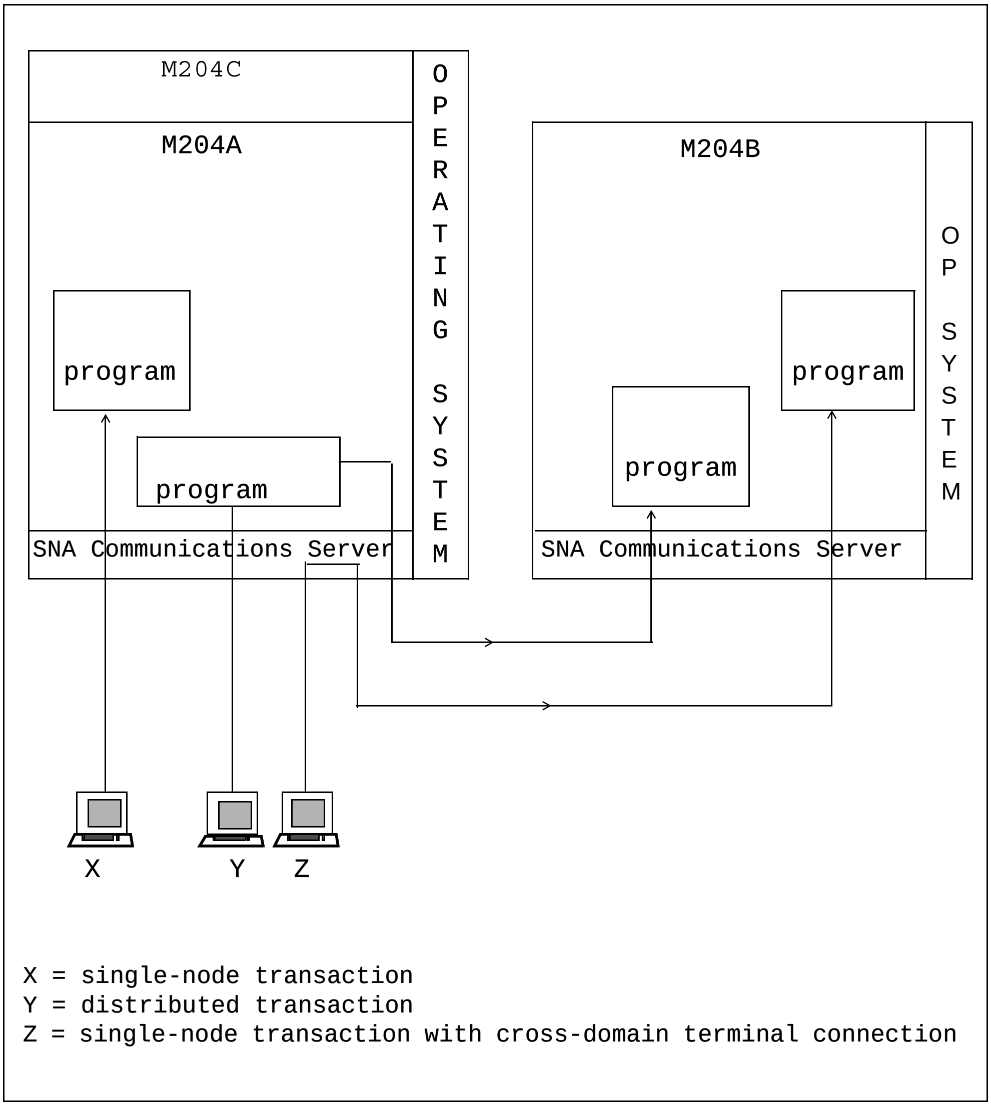

M204C
M204A
OPERATING SYSTEM
M204B
OP SYSTEM
program
program
program
program
SNA Communications Server
SNA Communications Server
X
Y
Z
X = single-node transaction
Y = distributed transaction
Z = single-node transaction with cross-domain terminal connection

Figure 2-1. Network Transactions

Essential elements of Figure 2-Figure 2-1. are described below:

- Each node on the network has a ***name*** by which it is known to other nodes. The node to the left is named M204A, and the one to the right is M204B.

- M204C is an additional node that runs on the same machine and in the same domain as M204A.

- Users X, Y, and Z are end users. An ***end user*** is defined as a user connected to a Model 204 terminal thread, such as IODEV 7 (SNA Communications Server 3270) or IODEV 41 (CMS Full Screen).

- User X is running a ***single node transaction***, that is, a transaction whose programs execute completely within a single application node of the network. This type of transaction has always been supported by Model 204.
- User Z is also running a single node transaction, but using a cross-domain terminal connection. User Z's terminal is owned by the SNA Communications Server on the machine that runs M204A, but is not involved with the M204A application node in any way. Z's terminal is connected to the application node M204B, which runs on a different machine (in SNA terms, "in a different domain"). This is still a single node transaction, because all the application programs involved run at a single application node: M204B.
- User Y is running a ***distributed transaction***. A distributed transaction is a transaction that requires application programs at more than one node. The exchange of data between two application programs at different nodes is referred to as a ***conversation***. Conversations are defined in greater detail in "SNA Concepts and Terminology," later in this chapter. The two programs that converse are referred to as ***conversation partners***.

Model 204's Intersystem Facilities

The Model 204 intersystem processing facilities, TPROCESS, Transfer Control, and Horizon, support distributed transactions. They are summarized below.

For more complete information about TPROCESS and Transfer Control, see:

- Rocket Model 204 documentation wiki SOUL pages: http://m204wiki.rocketsoftware.com/index.php/SOUL
- Rocket Model 204 documentation wiki system management pages: http://m204wiki.rocketsoftware.com/index.php/Using_Program_Communication_facilities
- Rocket Model 204 documentation wiki command pages: http://m204wiki.rocketsoftware.com/index.php/Category:Commands

Terminal Process Communication facility (TPROCESS)

This facility allows a SOUL (User Language) request to communicate with a partner process that runs in either CMS or CICS. The partner process is a CICS program or CMS EXEC. TPROCESS is a standard part of SOUL.

A listing of basic requirements, operation, and applications of TPROCESS follows.

Configuration requirements

The terminal user must connect to Model 204 through either the CICS or CMS full screen interface. SNA Communications Server is not required.

Operation and restrictions

- The partner process must run in the same physical machine as the SOUL request. The partner process runs in the environment (region or virtual machine) where the terminal is attached: CICS or CMS.
- The SOUL request must initiate the partner process. Under CICS, the request may transfer control to the partner.
- Communication passes through the terminal connection.
- The terminal thread IODEV number must be 11 or 41. For more information about these IODEVs, see the Rocket Model 204 documentation wiki:
  http://m204wiki.rocketsoftware.com/index.php/IODEV_parameter
- Under CICS, the partner process may be written in any supported host language. Under CMS, the language must be REXX or EXEC2.
- The SOUL request is always Master in the conversation, and the partner process is the Slave. This means that the SOUL request controls the communication channel, and that the partner may not send data on it unless the SOUL request is ready to receive it.

Applications

TPROCESS is appropriate for many applications that require interaction between a SOUL request and a program running in the terminal-owning environment. Examples include:

- Switching between a COBOL application running under CICS and a SOUL application running in Model 204.

  Control can be transferred between the two applications repeatedly without any awareness on the part of the end user that each application runs in a different environment.
- Passing DB2 or VSAM data from a CICS region into a Model 204 application.
- Editing a file using XEDIT under CMS, and then uploading the data into Model 204.
- Downloading data from Model 204 to CMS or CICS for manipulation by a statistical or graphics package.

Transfer Control facility

This facility allows a SOUL request to transfer control to a partner process that runs in CICS or to another Model 204 region or non-Model 204 SNA Communications Server application. The partner is a CICS program or SNA Communications Server application. Transfer Control is a standard part of SOUL.

A listing of basic requirements, operation, and applications of Transfer Control follows.

Configuration requirements

The terminal user must connect to Model 204 through either the CICS or SNA Communications Server full screen interface.

Only z/OS and z/VSE SNA Communications Server users can execute a transfer of control.

Operation and restrictions

- The partner can run in the same physical machine as the SOUL request or in a different physical machine: CICS partners must be in the same machine; SNA Communications Server partners can be in the same or in a different machine.
- The SOUL request transfers control to the partner.
- Communication passes through the terminal connection or through SNA Communications Server.
- For CICS partners, the terminal thread IODEV number must be 11. If the partner is a SNA Communications Server application, the terminal thread IODEV number must be 7. For more information about these IODEVs, see the Rocket Model 204 documentation wiki: http://m204wiki.rocketsoftware.com/index.php/IODEV_parameter

Applications

Transfer Control is appropriate for applications that require a transfer of a user from one Model 204 online into another or from one Model 204 online into a non-Model 204 CICS or SNA Communications Server application. Examples include:

- Under CICS, transferring a user back to an initiating transaction.
- Transferring a SNA Communications Server user between Model 204 regions without the user’s awareness of the change of environment.
- Testing Horizon applications: a programmer can transfer between Model 204 partners checking the results of a Horizon testing phase.

Horizon

This feature is more powerful and general than TPROCESS. It offers a communication interface in SOUL that conforms to the SNA LU 6.2 protocol. Horizon is an optional feature which must be purchased separately and must be installed to be used.

A listing of basic requirements, operation, and applications of Horizon follows.

Configuration requirements

The ACF/SNA Communications Server product or a supported TCP/IP product must be available in the operating system under which Model 204 is running. This guide assumes that the network product is SNA Communications Server.

TCP/IP support

Although SNA Communications Server is assumed, appropriate changes to the LINK and PROCESSGROUP command definitions allow the same Horizon code to work with a TCP/IP network product, unless mentioned otherwise in this guide.

On IBM z/VM and IBM z/VSE systems, Horizon supports IPv4 network addresses.

On IBM z/OS systems, Horizon supports either IPv4 or IPv6 network addresses. For full details on IPv6 network address support for Horizon, see the *Rocket Model 204 Release Notes: New Features Version 7 Release 4.0*.

Operation and restrictions

- The SOUL request's partner process may run in the same copy of Model 204, in an adjacent region or virtual machine, or in a different physical machine.
- The SOUL request may initiate the conversation, or the partner may initiate it.
- Communication passes through the SNA network (described in the next section, "SNA Concepts and Terminology").
- The terminal thread that initiates the conversation may be any IODEV number. The thread that receives an incoming conversation request must be IODEV 27. For more information about IODEV 27, see "IODEV specification for Horizon inbound threads" on page 44. See also the Rocket Model 204 documentation wiki system management pages: http://m204wiki.rocketsoftware.com/index.php/Defining_the_user_environment_(CCAIN).
- The partner process may be written in any language that has access to an LU 6.2 communication interface. Currently supported partner languages include:
  - SOUL
  - assembler
  - C
  - COBOL

  - PL/1
- The two communicating partners interact as peers. Each one can control the communication channel at different times during a conversation. Change of direction can be negotiated.

Applications

Horizon permits applications that require “any-to-any” connectivity. A transaction that originates in SOUL might request data from three partners, all of whom run on different hardware. This application can be designed so that the SOUL request has the same conversation with each partner, yet is completely unaware of the different hardware environments in which its partners run.

Horizon is therefore more flexible than TPROCESS, for example, which requires partners to run on the same physical machine. The relationship between the two Horizon partners is peer-to-peer, rather than the master/slave relationship required by TPROCESS.

Some possible applications include:

- A central site polling a number of satellites and aggregating data for a report.

  For example, a concern with several geographically distributed warehouses, each with its own IBM 9370-based inventory database, might want an aggregate total of its supply of a particular stock item.

- A central server used by several remote requestors.

  For example, an insurance company keeps premium calculation rules in a central database. Proposal generation is a distributed transaction: a branch office collects input data, ships it to the server for calculation, and then accepts the result and formats a report. One branch office keeps its local customer information in a Model 204 database on an IBM 9370, while a smaller office uses a Personal Computer.

- Aggregation of data from several different sources.

  A transaction interacts with an end user through a SOUL request, which invokes requests to CICS for DB2 data, to CMS for SQL/DS data, and even to a DEC computer for data residing there.

SNA Concepts and Terminology

This section provides a brief review of SNA concepts and how Horizon applies those concepts.

SNA networks provide resource sharing

Before the advent of Systems Network Architecture (SNA), data communication for IBM was mostly a matter of dedicated, hard-wired connections between communicating objects (usually a terminal communicating with a program). SNA moves toward an environment where network resources are managed by an operating system layer.

A major benefit of this approach is that network resources can be shared among different applications. An early example of this benefit is its removal of the need to dedicate a terminal to an application like CICS or Model 204. Instead, the terminal is “owned” by an operating system component called SNA Communications Server, which can connect it to CICS, Model 204, TSO, and so forth, at the terminal user’s request.

SNA protocols neutralize physical differences

A second objective of SNA is to standardize the rules, or protocols, by which two partners communicate. An important result of this is that partners have a logical, rather than a physical, view of each other. An application is not concerned about whether a terminal is attached locally or remotely, or about the idiosyncrasies of different terminal devices. At the application programming level, each communicating node in an SNA network is, therefore, referred to as a ***Logical Unit (LU)***, and each node on an SNA network is the point of connection for an LU.

Multiple LU types are defined; each is associated with the a set of rules (protocol) that the partners must obey to communicate. For inter-program communication, these rules specify an application program interface.

When this interface is offered by a particular product or language, it allows a program written in that language to converse with another program using a similar interface. Neither program needs any knowledge of what language its partner is written in, what hardware it runs on, what communication media are used, or what the physical network looks like.

This application program interface and the associated rules that specify how the communicating partners are to behave define LU type 6.2.

LUs service application programs

An LU is the system software under which an application program runs at a network node. With the installation of Horizon, Model 204 can become such an LU. Figure 2-2 on page 14 illustrates the relationship between an application program and an LU.

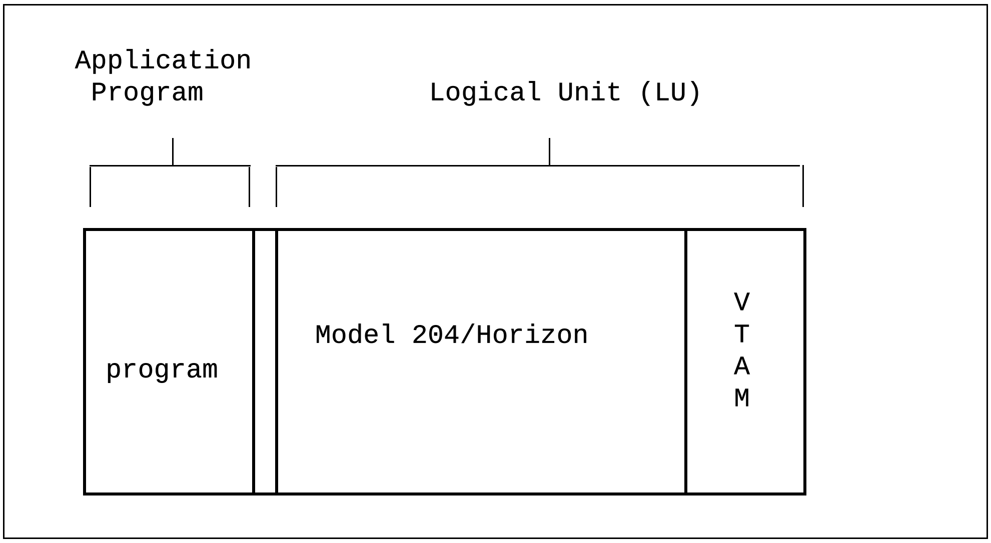


Figure 2-2. Application Program and LU

The LU provides various services to the application program:

- An application programming interface for communications.
- Enforcement of rules governing application program behavior.
- Packaging of application data in network format for transmission, and unpackaging and presentation to the application program of received application data.
- Error detection, and, when possible, recovery.

Horizon partner programs can be written in different languages

Horizon is Model 204’s implementation of LU 6.2. Using Horizon, a SOUL request can participate in a conversation with another program. That partner program can be another SOUL request or a program written in another language which satisfies the LU 6.2 conversation interface protocols.

For conversations where both partners are written in SOUL, the information in this Horizon guide is sufficient.

To design an application that involves conversations between programs in different languages, the designer must understand the common SNA standard to which the different interfaces conform. For example, the syntax of Horizon is somewhat different from that of the CICS interface and quite different from the subroutine-calling interface offered by APPC/PC. But all three LU 6.2 interfaces make a common set of operations available to an application program.

The architecture to which these and other LU 6.2 interfaces conform is documented in detail in IBM publication GC30-3084: the *Transaction*

Programmer's Reference Manual for LU Type 6.2. A mapping between Horizon's SOUL statements and their LU 6.2 architectural specifications is provided in Appendix B.

Sessions connect LUs; conversations connect programs

The connection between two logical units over which conversation data flows is called a ***session***.

Each session can be thought of as an individual telephone line between two LUs, in that it can be used by only one pair of conversing programs at any given time. A session is thus a resource "owned" by an LU and assigned to an application program for the duration of a conversation. A session between two LUs usually survives after a conversation ends and is then free to be reassigned to a subsequent conversation.

A ***session limit*** can be set between any two LUs. This limit determines how many concurrent conversations can be active between them; it puts a cap on the number of parallel sessions that may exist between the two LUs.

A ***conversation*** is a communication between two application programs. The program that initiates the conversation is called the ***client*** program. The client's partner in the conversation is called the ***server*** program. When the LU at which the server program is to run receives the initiation request, it starts the server program.

A session can be thought of as a long-term connection between two LUs, whereas a conversation is seen as a short-term connection between two application programs. Since a conversation flows on a session, a session must be established, or ***bound***, between the LUs involved before two application programs can have a conversation.

Horizon supports parallel sessions

A pair of LUs can connect to each other by a ***single session*** (one session active at a time, reused serially for conversations) or by ***parallel sessions*** (multiple sessions active at a time). Two LUs cannot be connected by both single and parallel sessions at the same time.

Parallel session support allows the partitioning of sessions into session groups that have common properties such as the remote partner and session limit and common session characteristics like message size and security.

Parallel sessions are supported in Model 204-to-Model 204 communications. Model 204 also provides CNOS (change number of sessions) parallel session support, and can connect with parallel sessions to any client that provides the CNOS minimum support set of functions and components (as defined in the *SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2)*.

For parallel sessions connections from Model 204 to systems other than those running on SNA Communications Server, Model 204 must run with the so-called “peer-to-peer” release levels of SNA Communications Server and NCP.

Conversation partners are known as client and server

Figure 2-Figure 2-3. illustrates a distributed transaction and facilitates discussion of the relationship between the two conversation partners.

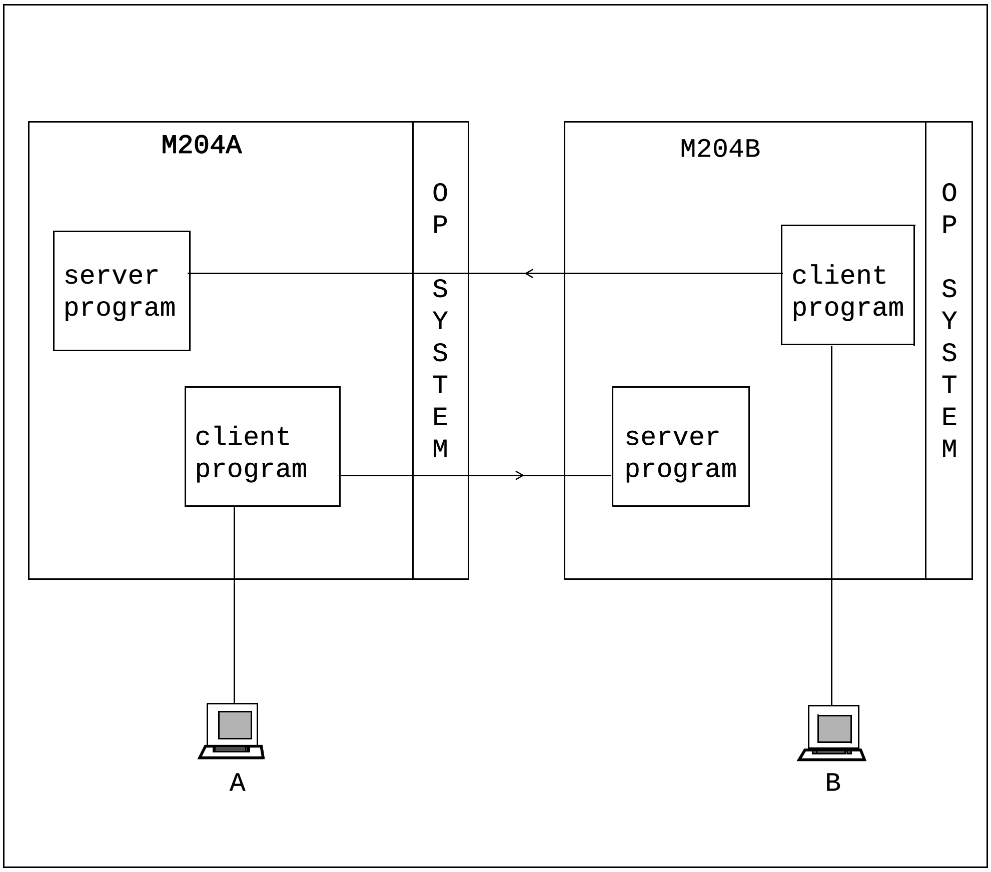


Figure 2-3. Horizon Distributed Transaction

The end user sits at a terminal. The thread that the end user is connected to runs the client program. The client program, also referred to as the ***outbound*** program, makes the initial request to establish a conversation, using the OPEN PROCESS statement.

The client program’s OPEN PROCESS statement causes a conversation initiation request to pass over the communication path. The LU at which the server program is to run receives the initiation request and starts the server program. The server program, which services the request for a conversation, begins by issuing the OPEN PROCESS ACCEPT statement. The server program is also referred to as the ***inbound*** program.

A Model 204 system in which a client program runs is referred to as a ***client system***; one in which a server program runs is called a ***server system***.

A system is designated client or server only in the context of a particular transaction. In Figure 2-3 on page 16 ONLINE M204A is the client system for transaction A, which initiates at M204A. For transaction B, which initiates at M204B, M204A is the server system.

Note: A client program never connects to an active server program. The OPEN PROCESS request actually initiates, or attaches, the server program. For Model 204, this implies that there must be a free (that is, not currently used) thread on the server system to run the server program. This free thread must be a Horizon IODEV (IODEV=27), defined by the system manager in the user zero input stream.

For more information about the Horizon IODEV, see “IODEV specification for Horizon inbound threads” on page 44. See also the Rocket Model 204 documentation wiki system management pages: http://m204wiki.rocketsoftware.com/index.php/Defining_the_user_environment_(CCAIN).

Horizon programs can have multiple roles and partners

As shown in Figure 2-Figure 2-4., it is possible to build distributed transactions that span more than two nodes.

In Figure 2-Figure 2-4., the application program at M204A has two open conversations: one with a program at the M204B node, and one at M204C. The application program at M204B is both a server to the client program at M204A, and a client program to the server at M204D.

The designation of a program as client or server carries no implications about how it behaves with respect to its partner once the conversation is established. These terms only describe who initiated the conversation (the client), and who was initiated by it (the server). Since the Horizon protocol is peer-to-peer, which partner has control of the conversation (that is, the one allowed to send data at any given point in processing) is completely up to the application designer.

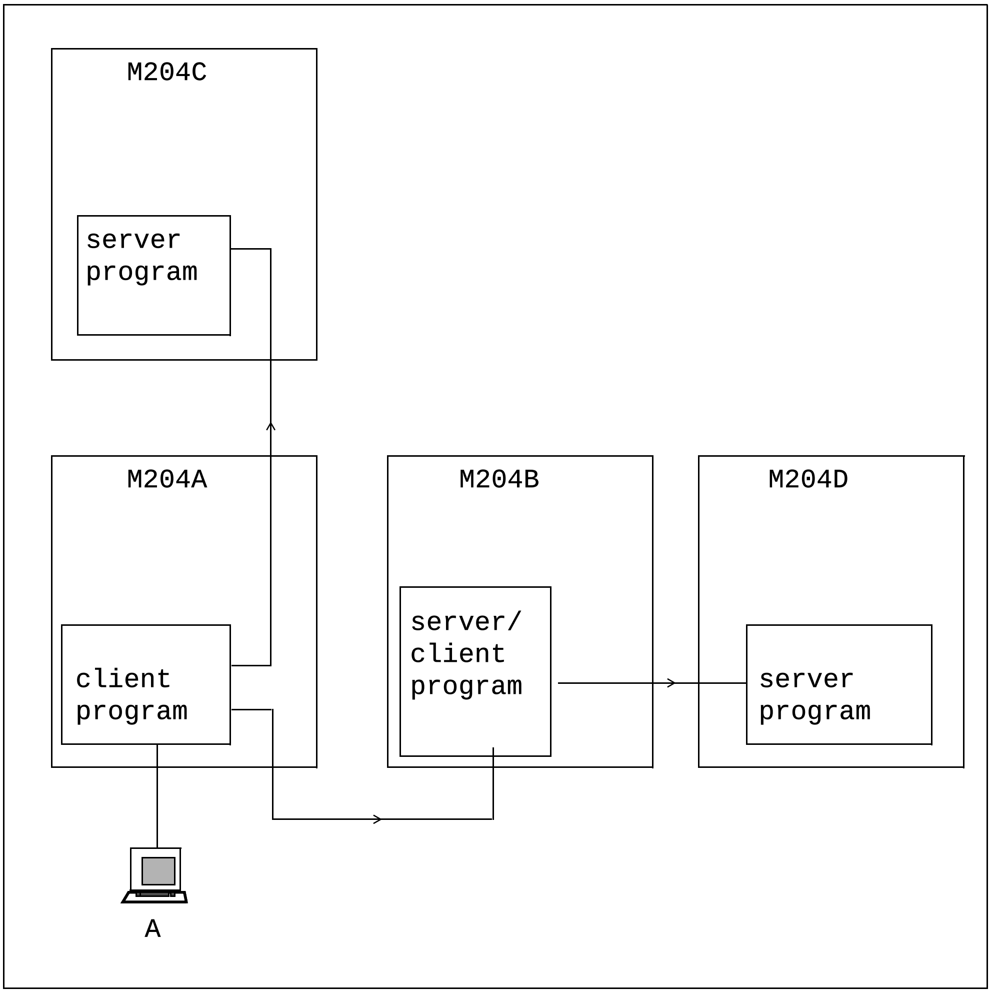


Figure 2-4. Multi-Node Transaction

Horizon terms have multiple synonyms

Unmodified, “client” can refer to a Model 204 ONLINE system or to an application program that runs in that Model 204 ONLINE. “Server” similarly can refer to a system or a program. To clarify the terms used in this document, the Table 2-1 lists possible elements in a typical Horizon network transaction along with their synonyms and definitions.

Table 2-1. Synonyms for Horizon terms

Term	Also known as	Meaning
end user	terminal user	User on a terminal thread connected to the client system

Table 2-1. Synonyms for Horizon terms (Continued)

Term	Also known as	Meaning
client program	[client | outbound} {user | program | process | application	Model 204 application that initiates a conversation
client system	{client | outbound} {system | Online | node | LU}	Host and provider of network services to a client program
server program	{server | service | inbound} {user | program | process | application}	Model 204 application that is initiated by an incoming conversation request.
server system	{server | service | inbound} {system | Online | node | LU}.	Host and provider of network services to a server program

3

Network Management

Overview

The network management topics that the system manager must consider when using Model 204's Horizon are listed below. This chapter discusses each of these topics except for network security, which is described in Chapter 4.

- Network definition

  Network definition involves defining the network to Model 204 (using the DEFINE command) and defining the network to SNA Communications Server (formerly VTAM) or a TCP/IP product.

- Network administration

  Network administration involves using the network administration commands to manipulate the network entities defined to Model 204.

- Network security

  Network security involves using DEFINE command security parameters in conjunction with Application Subsystem security to provide the desired protection for network resources. Network security is described in Chapter 4, "Security."

- Model 204 Online configuration preparation

  Online configuration preparation involves the following modification of system input (CCAIN): allotting Horizon inbound threads (IODEV=27) and adding to the NSUBTSKS parameter value for LU 6.2/SNA Communications Server pseudo subtasks.

- Application program testing

The Remote Procedure Invocation (RPI) subsystem, included as part of DICTIONARY installation, is an option that simplifies the testing of Horizon application programs.

Defining the Network to Model 204

Before partner programs can converse over Horizon, the network, including the programs and their interconnections, must be defined to Model 204. In addition, the network must be defined to SNA Communications Server, which supports Horizon. Each Model 204 system in the network provides a definition of the link, processgroup, and process entities using respectively the DEFINE LINK, DEFINE PROCESS, and DEFINE PROCESSGROUP commands.

Defining the network to	Is described in
Model 204	• This chapter. • The DEFINE LINK, DEFINE PROCESS, and DEFINE PROCESSGROUP commands on the Rocket Model 204 documentation wiki: http://m204wiki.rocketsoftware.com/index.php/Category:Commands
SNA Communications Server	"Defining the Network to SNA Communications Server" on page 36.

Network entities requiring definition

This section introduces the network definition commands. For Model 204, Horizon network definition involves identifying network entities with DEFINE commands. These entities and their interdependencies form the network's logical layer. Parameters of the DEFINE command for each entity point to names of related entities within the network and thereby determine the network's communication paths.

The following three network entities must be defined with the Model 204 DEFINE command before a Horizon application can run successfully:

- Link — Defines the transport mechanism and protocol to be used for communications, and specifies a node name to be used when Model 204 connects to the SNA network so that other nodes within the network can refer to this node by name.
- Processgroup — Connects the process to a specific link and groups processes according to certain attributes, such as session usage, to facilitate resource allocation. It also defines a remote node with which a process can communicate.
- Process — Defines the local conversation program. If the process is a client process, it also identifies the remote conversation partner or partner process. If the process is a server process, it identifies the subsystem which is to be invoked when the client process issues an inbound conversation request.

Note: CNOS connections require the definition of two additional definition commands: DEFINE REMOTE and DEFINE SESSIONGROUP. For more information about these entities, see Appendix E.

Also, for the complete syntax and description of DEFINE LINK, DEFINE PROCESSGROUP, DEFINE PROCESS, DEFINE REMOTE, and DEFINE SESSIONGROUP, see the Rocket Model 204 documentation wiki:

http://m204wiki.rocketsoftware.com/index.php/Category:Commands

Creating a Horizon network

A large department store chain has its headquarters in New York and branches located in various states in the U.S. Three Model 204 online systems are running in three different locations: the headquarters in New York, one branch in San Francisco, and one branch in Boston. Figure 3-1 shows the network to be defined to Model 204.

Polling application with one-way communication

Figure 3-1 illustrates a polling application where communication is initiated in only one way: from headquarters to the two branches. Headquarters needs to access the weekly sales figures in San Francisco and Boston.

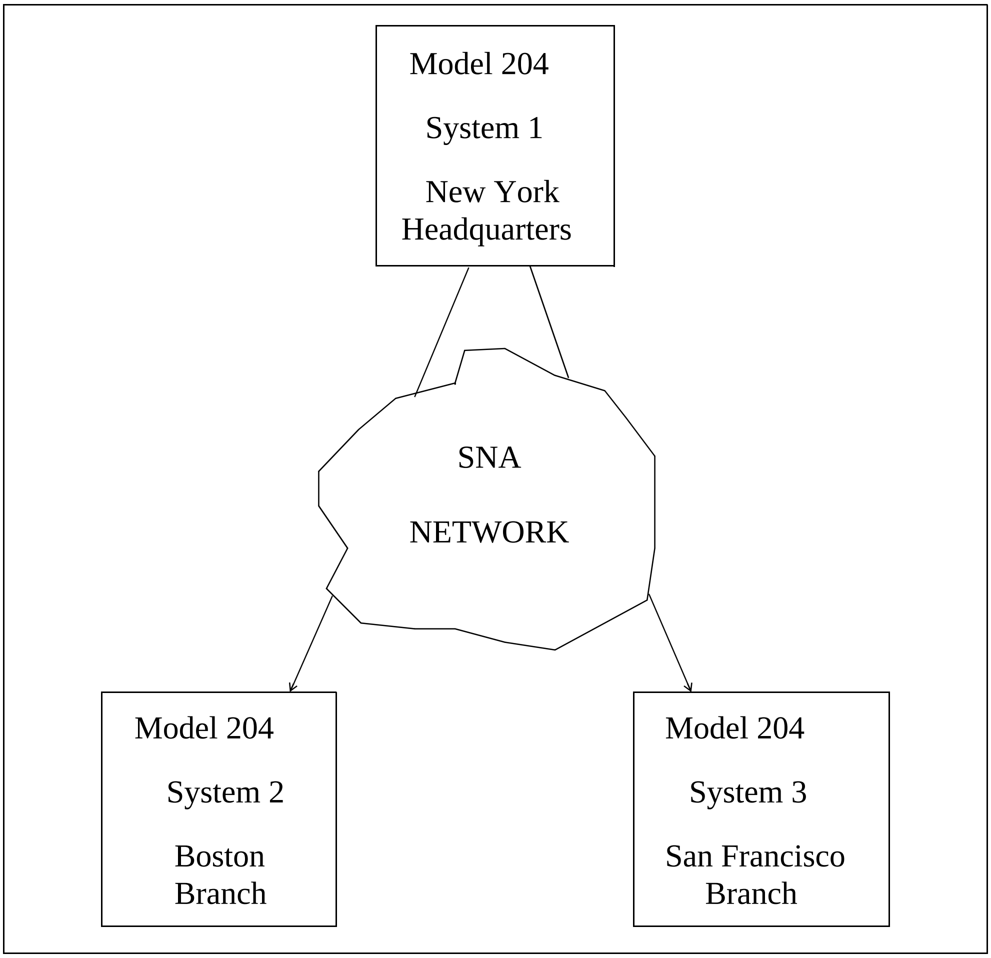


Figure 3-1 Physical Network Configuration for Polling Application

DEFINE parameters for network entities

Figure 3-2 on page 25 illustrates the logical view from the Model 204 System 1 at headquarters and at the Boston Branch, Systems 1 and 2 respectively. The boxes contain skeleton entity definitions: named entities plus the appropriate DEFINE command parameters. The arrows in Figure 3-2 show how the parameters interrelate: straight arrows show intrasystem correspondence; bent arrows show intersystem correspondence. The complete formats of the DEFINE commands are described in the Rocket Model 204 documentation wiki:

http://m204wiki.rocketsoftware.com/index.php/Category:Commands

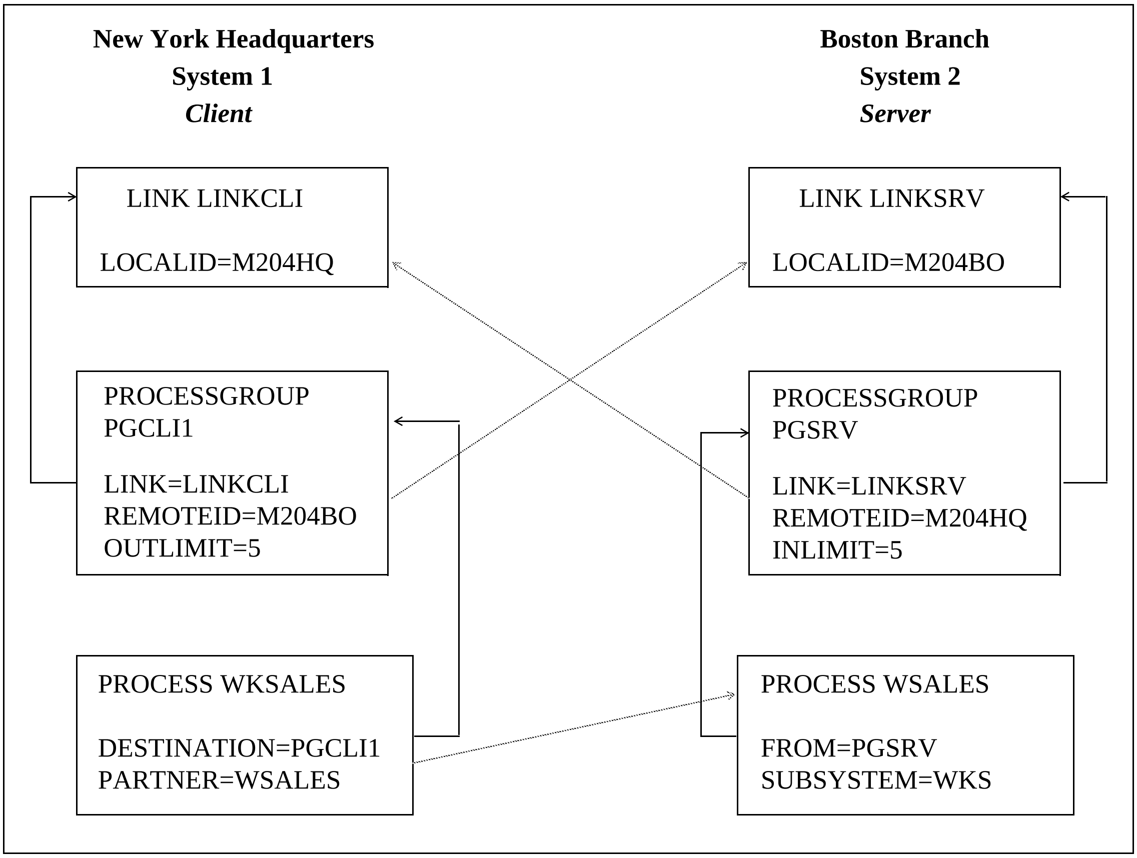


Figure 3-2 Logical Network View from New York and Boston

Defining the link

Each Model 204 system is identified to the network through the LOCALID parameter of the DEFINE LINK command. In this example, the LOCALID for the New York headquarters is M204HQ; the LOCALID for the Boston Branch is M204BO.

Defining the processgroup

The DEFINE PROCESSGROUP command uses the REMOTEID parameter to specify the remote node with which a process wishes to communicate. Processgroup PGCLI1 in headquarters specifies REMOTEID M204BO, which points to the Boston branch; processgroup PGSRV in the Boston branch specifies REMOTEID M204HQ, which points to the New York headquarters.

Defining the process

The DEFINE PROCESS command for the client process in headquarters specifies PARTNER=WSALES, the server process in Boston. The server process WSALES in Boston specifies SUBSYSTEM=WKS, the subsystem to be invoked when the client process initiates a conversation.

Defining a Horizon network

Example 1: Polling Application

The following is a step-by-step description of how the DEFINE commands are set up for a polling application where the conversations are initiated from one central location. This example is based on the department store application described in the physical network example in Figure 3-1 on page 24. DEFINE commands for the headquarters in New York are described first.

Defining the link for the client side

New York headquarters initiates a conversation to determine the weekly sales figures at the Boston and San Francisco branches. New York headquarters is therefore the client system. The first command required is DEFINE LINK:

```
DEFINE LINK LINKCLI WITH  -
SCOPE=SYSTEM              -
TRANSPORT=VTAM            -
PROTOCOL=LU62             -
SESSIONS=6                -
LOCALID=M204HQ            -
INBUFSIZE=2048
```

This definition provides the following specifications:

- The transport type (SNA Communications Server) and the protocol (LU62) to be used for communications.
- The maximum number of sessions that can be used by Horizon applications (6).
- The identification of the New York headquarters' Model 204 system to the SNA network (M204HQ).
- The size of the "receive" buffer, which receives data transferred over the session (2048 bytes, the recommended value).

Defining the processgroup for the client side

The next step is setting up the processgroup definitions:

```
DEFINE  PROCESSGROUP PGCLI1 WITH   -
        SCOPE=SYSTEM               -
        LINK=LINKCLI               -
        REMOTEID=M204BO            -
        OUTLIMIT=5                 -
        INLIMIT=0                  -
        RETAIN=4
```

```
DEFINE  PROCESSGROUP PGCLI2 WITH   -
        SCOPE=SYSTEM               -
        LINK=LINKCLI               -
        REMOTEID=M204SF            -
        OUTLIMIT=3                 -
        INLIMIT=0                  -
        RETAIN=0
```

This definition provides the following specifications:

- The first processgroup defined above, PGCLI1, is to converse with the Model 204 system at the Boston branch (REMOTEID=M204BO).
- PGCLI1 allows a maximum of five concurrent outbound conversations (OUTLIMIT=5).
- Since the headquarters is initiating all the conversations and is not expected to receive any inbound conversation requests, no inbound conversations have to be specified (INLIMIT=0).
- Four sessions are retained.
- The second processgroup, PGCLI2, is to converse with the Model 204 system at the San Francisco branch.
- PGCLI2 allows only three concurrent outbound conversations.
- Since the headquarters is initiating all conversations and is not expected to receive any inbound conversation requests, no inbound conversations have to be specified.
- No sessions are retained when processes belonging to PGCLI2 are closed.

Defining the client process

Lastly, the system manager defines the remote conversation partner with the DEFINE PROCESS command as follows:

```
DEFINE  PROCESS WKSALES WITH                -
        SCOPE=SYSTEM                        -
        DESTINATION=(PGCLI1,BOSTON,PGCLI2,SF) -
        PARTNER=WSALES                      -
        DATALEN=2048                        -
        CONFIRM
```

The definition provides the following specifications:

- Process WKSALES belongs to processgroups PGCLI1 and PGCLI2, which directs the conversation to either the Boston or San Francisco branches.
- WKSALES is a client process and its partner name is WSALES, the process name on the server system.
- The maximum data area expected to be received from the server processes on either the Boston or San Francisco branches is 2048 bytes.

- Confirmation processing is allowed.

Logically viewing the network

The network as seen by the system manager at the New York headquarters is shown in Figure 3-3.

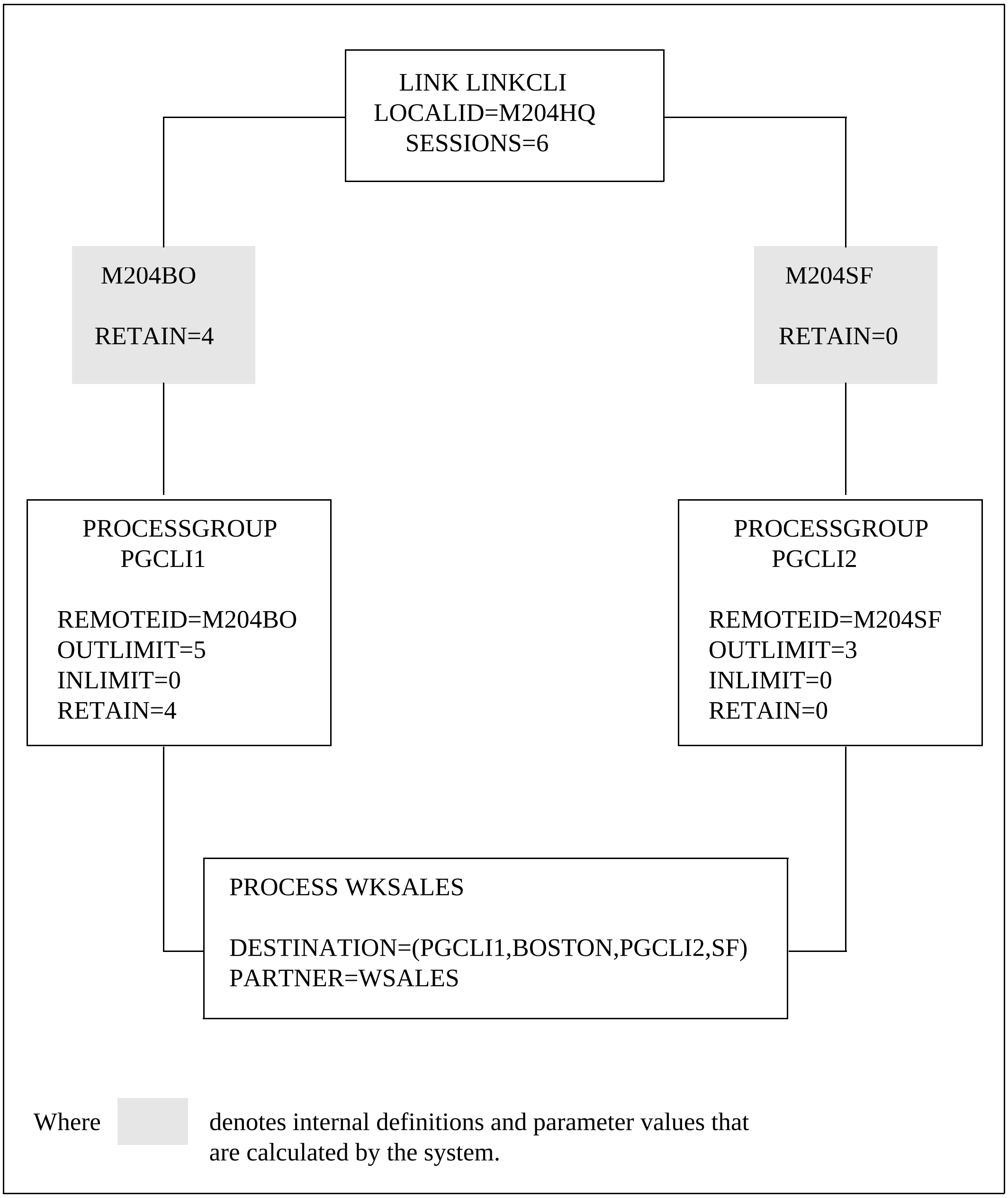


Figure 3-3 Logical Network Configuration at New York Headquarters

Figure 3-3 shows the relationship of the process to the processgroups and the processgroups to the link. Notice that the two processgroups support a total of eight concurrent outbound conversations, while the link supports only six sessions for these outbound conversations.

PGCLI2 is assumed to be a less active processgroup that requires fewer and shorter conversations. Its RETAIN value of zero means that when its conversations end, its sessions are freed up for use by the busier PGCLI1.

Preserving an available session

This is an example of tuning RETAIN and OUTLIMIT values so that it is possible not to run out of available sessions. See the discussion of RETAIN in “Defining a Horizon network” on page 26 for more information about sharing sessions and network resource use.

The sessions are allocated on a first come, first served basis. They remain allocated to a remote node unless the link is closed, a STOP LINK or STOP PROCESSGROUP command is issued, or the session is ended by the remote node. The effects of STOP LINK and STOP PROCESSGROUP are described further in the Rocket Model 204 documentation wiki:

http://m204wiki.rocketsoftware.com/index.php/STOP_command:_Stopping_a_Horizon_or_Model_204_SQL_link_or_processgroup

The session used by each outbound conversation with PGCLI2 is released as soon as the conversation ends, as specified by the RETAIN value of zero in PGCLI2’s processgroup definition.

Defining the link for the server side

Since the branches receive inbound conversation requests from the New York headquarters for weekly sales figures, they are server systems. The DEFINE commands for one of the branches, Boston, are as follows, beginning with DEFINE LINK:

```
DEFINE LINK LINKSRV WITH  -
       SCOPE=SYSTEM       -
       TRANSPORT=VTAM     -
       PROTOCOL=LU62      -
       SESSIONS=5         -
       LOCALID=M204BO     -
      INBUFSIZE=2048
```

This definition provides the following specifications:

- The transport type (SNA Communications Server) and the protocol (LU62) to be used for communications.
- The maximum number of sessions that can be used by Horizon applications (5).
- The identification of the Boston branch’s Model 204 system to the SNA network (M204BO).

- The size of the “receive” buffer, which receives data transferred over the session (2048 bytes, the recommended value).

Defining the processgroup for the server side

The next step is setting up the processgroup definition:

```
DEFINE PROCESSGROUP PGSRV WITH  -
       SCOPE=SYSTEM             -
       LINK=LINKSRV             -
       REMOTEID=M204HQ          -
       OUTLIMIT=0               -
       INLIMIT=5                -
       RETAINALL
```

The definition provides the following specifications:

- Processgroup PGSRV needs to converse with the Model 204 system at the New York headquarters, as indicated by the REMOTEID value of M204HQ.
- PGSRV allows a maximum of five concurrent inbound conversations and no outbound conversations, since the Boston branch is not expected to initiate any conversations.
- All sessions are retained.

Defining the server process

Lastly, the DEFINE PROCESS command specifies the subsystem to be invoked by an inbound conversation request from the client:

```
DEFINE  PROCESS WSALES WITH  -
        SCOPE=SYSTEM         -
        FROM=PGSRV           -
        SUBSYSTEM=WKS        -
        DATALEN=2048         -
        CONFIRM
```

The definition provides the following specifications:

- Process WSALES belongs to processgroup PGSRV, which directs the conversation to the New York headquarters.
- WSALES is a server process and the subsystem associated with it is WKS.
- The maximum data area expected to be received from the client process is 2048 bytes.
- Confirmation processing is allowed. Even if the SOUL CONFIRM statement is not used in the subsystem procedures, CONFIRM is included in the server process definition because the client process has CONFIRM in its definition. Both the client and server processes must specify the same option.

Example 2: Polling Application Extension

The following example highlights the RETAIN parameter of the DEFINE PROCESSGROUP command. The department store application described in the previous example is expanded here, requiring more than weekly sales figures from the Boston and San Francisco branches. Suppose that the New York headquarters also requires the payroll and personnel information at the branches in order to print paychecks on a monthly basis.

Logical view of network

The logical view of the network at New York headquarters is illustrated in Figure 3-4. The system manager uses two different processgroups to handle sales and payroll applications. This limits the maximum number of concurrent outbound conversations for each type of application.

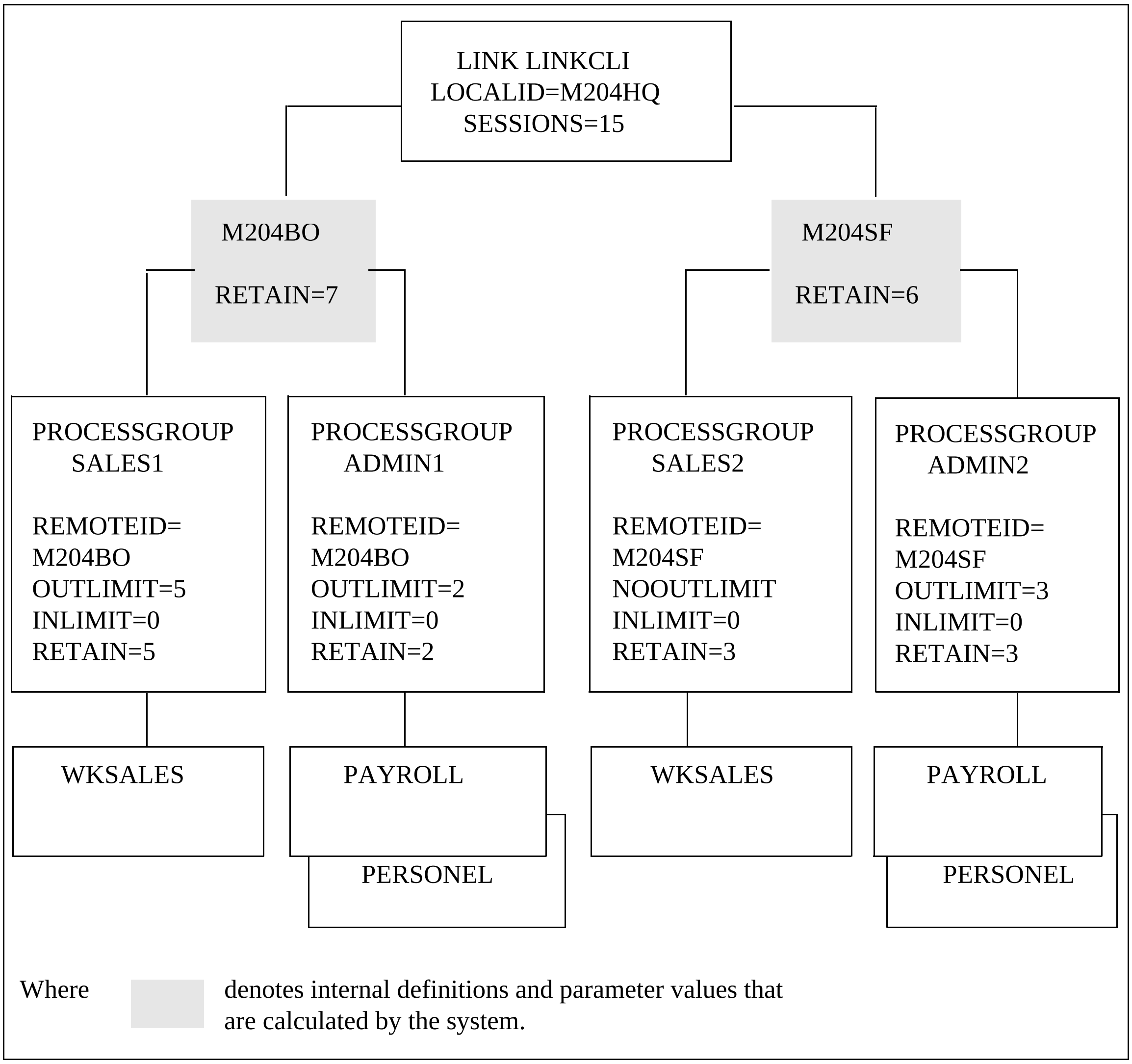


Figure 3-4 Processgroups Sharing Sessions

Processgroups sharing sessions

Processgroups SALES1 and ADMIN1 (and SALES2 and ADMIN2) are able to share sessions because of four parameter values they have in common:

- LINK=LINKCLI
- REMOTEID=M204BO
- LOGIN=NOTRUST (the default)
- MODENAME (not specified)

Processgroups SALES1 and ADMIN1 can retain a total of seven sessions since SALES1 is specified with RETAIN=5 and ADMIN1 is specified with RETAIN=2.

Note: The sessions are shared by SALES1 and ADMIN1. This does not mean that five sessions are retained specifically for SALES1 and two for ADMIN1.

Processgroups SALES2 and ADMIN2 can retain a total of six sessions since SALES2 is specified with RETAIN=3 and ADMIN2 is specified with RETAIN=3. Since SALES2 has NOOUTLIMIT, it is possible for all six sessions to be used by processes belonging to SALES2.

Example 3: Two-Way Conversation Initiation Application

The polling application in Figure 3-1 on page 24 involved communication initiated in only one way: from one central location, the headquarters, to the two main branches. Communication can also be initiated from more than one direction.

For example, the Boston branch mentioned in the previous example may require access to the inventory in a sub-branch located in Cambridge. If the Boston branch runs out of stock on an item, it can be determined whether Cambridge has the item available, and the customer can be sent to that branch. Similarly, the Cambridge branch can do a query on the Boston branch’s inventory.

Physical network configuration

Figure 3-5 shows the physical network configuration for this type of application.

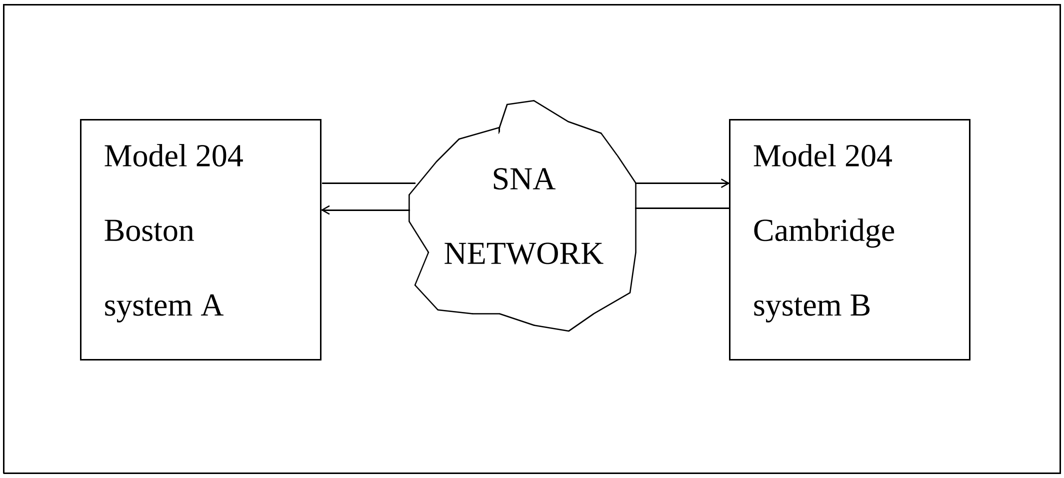


Figure 3-5 Two-Way Conversation Initiation

Logical network view

Figure 3-6 on page 34 shows the logical network configuration for the two-way inventory application as viewed from the Boston branch. Processgroup INVENTRY points to M204CB, which identifies the Cambridge branch. Both the OUTLIMIT and INLIMIT parameters for processgroup INVENTRY are set to nonzero values: it is possible to initiate conversations or receive inbound conversation requests.

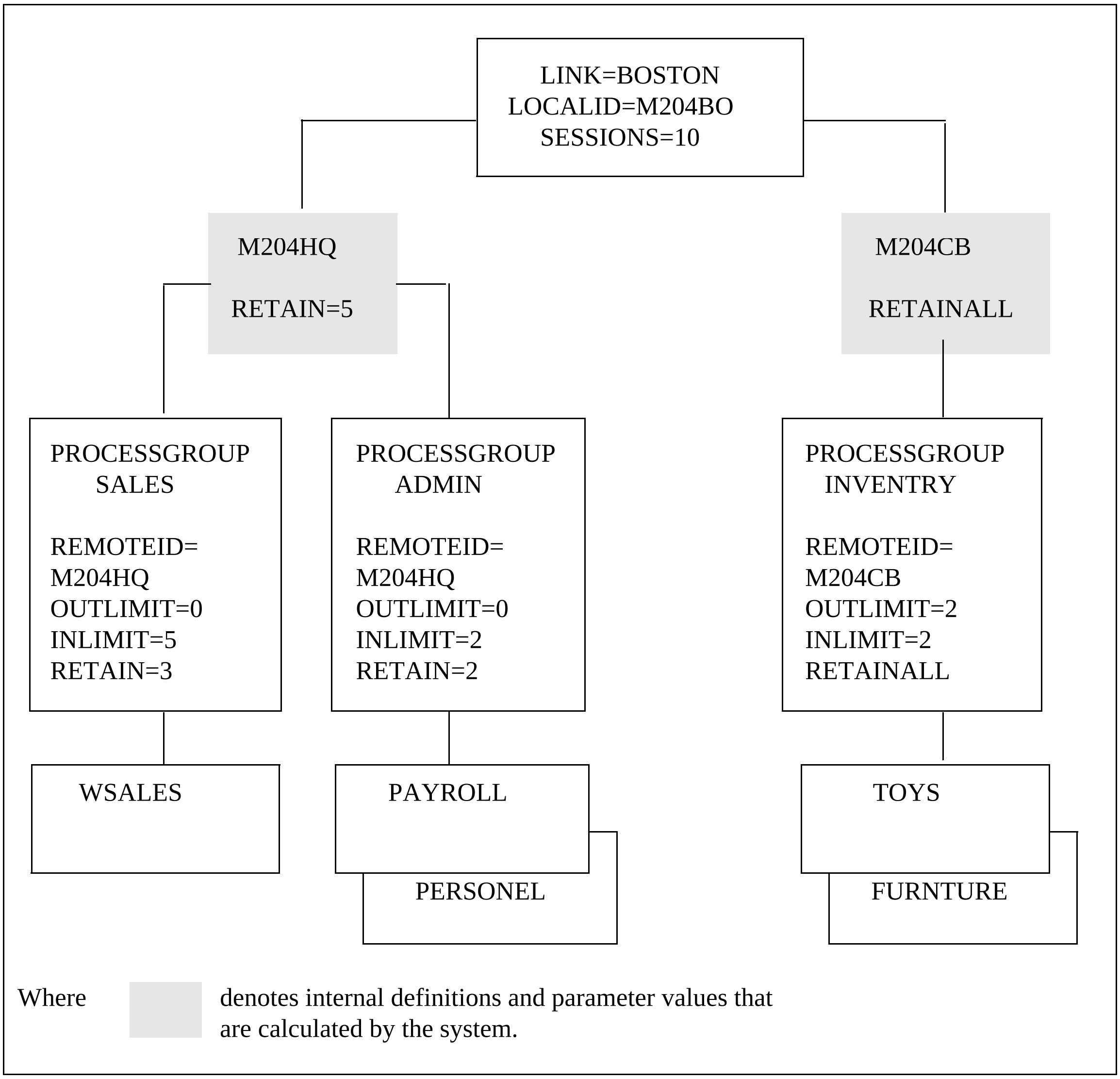


Figure 3-6 Logical Network Configuration for Boston Branch

Example 4: Local Node Testing

A single Model 204 system can represent both the client and the server system for initial development and testing purposes. You can set up two SNA Communications Server APPL definitions for the same Model 204 ONLINE, and use two different link definitions to refer to these APPL definitions using the LOCALID parameter. For more information about SNA Communications Server APPL definitions, see “Defining the Network to SNA Communications Server” on page 36.

Logical network view

Figure 3-7 on page 35 shows the interrelationship of the network entities for a single Model 204 system acting as client and server. The entity names must be unique since they reside within the same Model 204 online system. The arrows in the figure show how the DEFINE command parameters interrelate: straight line arrows show intrasystem correspondence; bent line arrows show intersystem correspondence.

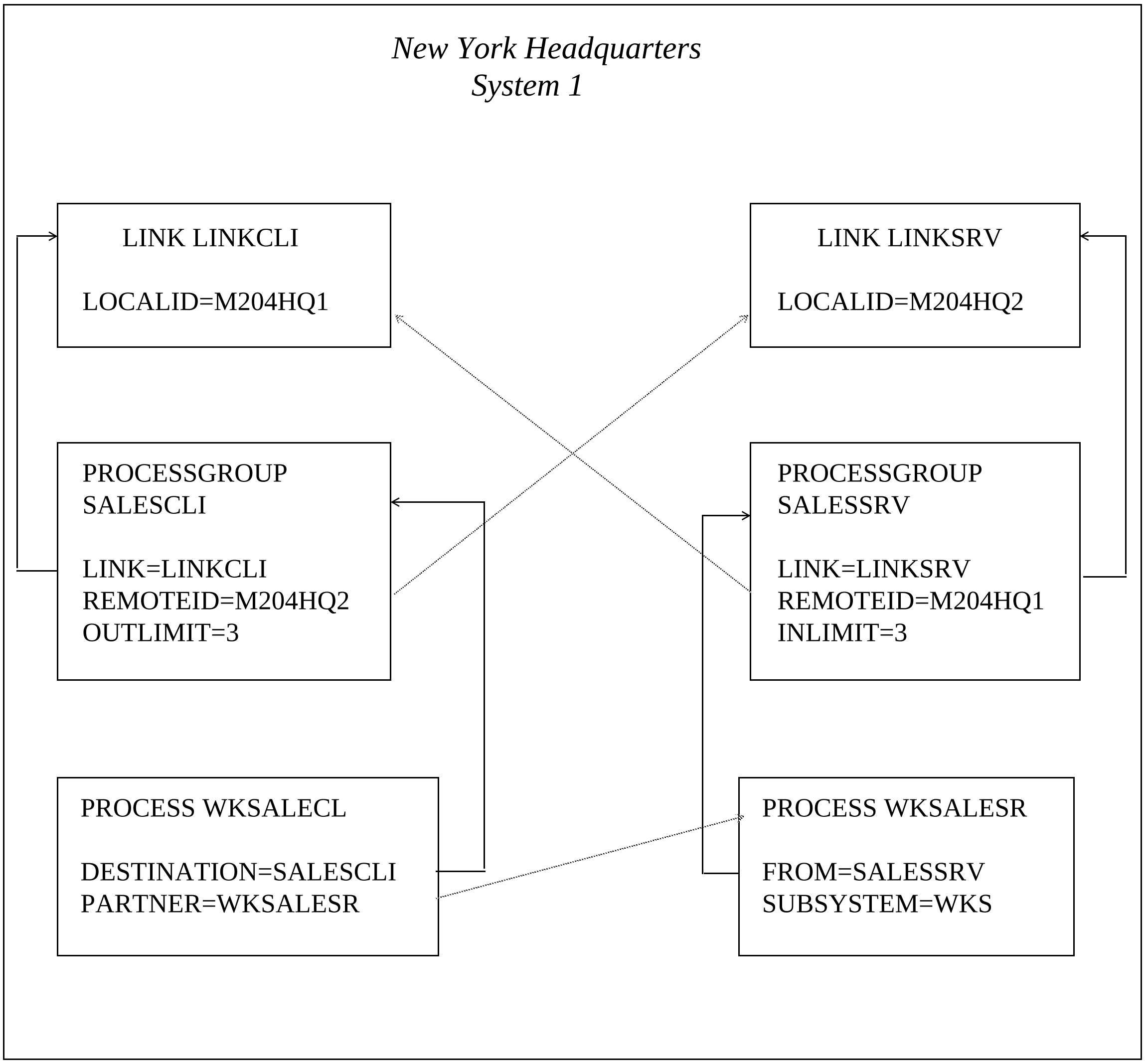


Figure 3-7 Client and Server Processes on a Single System

Defining the links

```
DEFINE LINK LINKCLI WITH    -
       SCOPE=SYSTEM         -
       TRANSPORT=VTAM       -
       PROTOCOL=LU62        -
       SESSIONS=5           -
       INBUFSIZE=2048       -
       LOCALID=M204HQ1

DEFINE LINK LINKSRV WITH    -
       SCOPE=SYSTEM         -
       TRANSPORT=VTAM       -
       PROTOCOL=LU62        -
       SESSIONS=5           -
       INBUFSIZE=2048       -
       LOCALID=M204HQ2
```

Defining the processgroups

```
DEFINE PROCESSGROUP SALESCLI WITH  -
       SCOPE=SYSTEM                -
       LINK=LINKCLI                -
       REMOTEID=M204HQ2            -
       OUTLIMIT=3                  -
       INLIMIT=0                   -
       RETAIN=3

DEFINE PROCESSGROUP SALESSRV WITH  -
       SCOPE=SYSTEM                -
       LINK=LINKSRV                -
       REMOTEID=M204HQ1            -
       OUTLIMIT=0                  -
       INLIMIT=3                   -
       RETAIN=3
```

Defining the processes

```
DEFINE PROCESS WKSALECL WITH       -
       SCOPE=SYSTEM                -
       DESTINATION=SALESCLI        -
       PARTNER=WKSALESR            -
       DATALEN=2048                -
       IMEOUT=60

DEFINE PROCESS WKSALESR WITH       -
       SCOPE=SYSTEM                -
       FROM=SALESSRV               -
       SUBSYSTEM=WKS               -
       DATALEN=2048                -
       TIMEOUT=60
```

Defining the Network to SNA Communications Server

This section describes how to set up the SNA Communications Server network support for Horizon.

Throughout the discussion in this section, the term link refers to a Horizon link. Although not a communication line like a SNA Communications Server or SNA link, a Horizon link, in fact, corresponds to an SNA logical unit (LU). That is, a

Horizon link serves as a *logical port* between an application program and the network.

Two components of SNA Communications Server network definition

You define Horizon to the SNA Communications Server network in two areas of the VTAMLST data set:

1. APPL definition statements
2. Mode table entries

An APPL statement for each Horizon “link” is required to establish that Horizon link as a logical unit (LU) within the network.

An entry in a mode table is required to specify the particular set of SNA protocols (the session mode) desired for a session between two Horizon LUs. In this case the protocols are characteristic of LU type 6.2.

The following two sections describe how to code the APPL statements and the mode table entries to define the SNA Communications Server network.

APPL Statements for Network Definition

SNA Communications Server network definition for host-to-host communications (such as a Model 204-to-Model 204 Horizon conversation) differs from that for terminal-to-host environments. In host-to-host environments, a single entity type can take either the primary or the secondary role in the conversation. Terminal-to-host environments, have two entity types, one of which always takes the primary role and the other of which always takes the secondary role.

APPL statement per Horizon online

Horizon Model 204-to-Model 204 sessions involve one type of LU: a SNA Communications Server application LU. Terminal-to-host communications involve two different types of LUs: a SNA Communications Server application LU and several terminal LUs. These LU types are defined in SNA Communications Server in separate places. Host Onlines (SNA Communications Server application LUs) are defined by APPL statements; terminals are defined in LU macros.

APPL statement dual function

In a Horizon Model 204-to-Model 204 session, both partners are SNA Communications Server application LUs and both are defined to SNA Communications Server by APPL statements. Yet, on any particular Horizon session, one partner is the primary end and the other one is the secondary, just as in a terminal-to-host session. The APPL statement for a Horizon system must therefore provide information about both the primary and secondary ends.

In terminal-to-host systems the APPL statement carries the primary information and the LU macros carry the secondary information.

Thus, for example, the APPL statement for a Horizon system must point to a mode table (while for terminal-to-host systems the LU macro does). Also, since each Horizon link supports multiple sessions, where the link can be primary on one and secondary on another, information for both roles must be included on each APPL.

Coding the APPL statement(s) for Horizon support

The APPL statements are coded within the library SYS1.VTAMLST (in z/OS) or within a file with filetype VTAMLST (in CMS).

You should group together all the APPL statements for all the Model 204 systems running under a particular SNA Communications Server in the same VTAMLST member or file, for ease of activating the APPL statements.

How many APPL statements are required?

In addition to the APPL statement required for Model 204 SNA Communications Server terminal support, you must code at least one APPL statement for each Model 204 system involved in a Horizon conversation.

Each additional link to the network requires an additional APPL statement.

Activation of APPL statements

For each SNA Communications Server that supports Horizon “links,” the relevant APPL statements must be activated before Model 204 and Horizon can use the network. Generally, this activation occurs automatically at SNA Communications Server startup. However, if direct operator control of APPL activation is desired, entire groups of APPLs (major nodes) can be activated at one time.

APPL statement parameters

Most of the two dozen or so parameters on the APPL statement default to values appropriate to Horizon systems. Only four parameters must be specified by the SNA Communications Server systems programmer. Technical Support recommends that two other parameters be specified as well, to make network maintenance easier. A seventh parameter, again optional, is relevant only where Horizon node name protection is desired. Node name protection is described on “Protecting Network Node Names” on page 60.

The required parameters for the APPL statement, when used for defining SNA Communications Server support for Horizon links, are shown in the following simplified syntax chart that does not include all APPL statement parameters.

```
name APPL  AUTH=ACQ,
                PARSESS=YES,
                MODETAB=tablename,

Optional:
                DLOGMOD=entryname,
                VPACING=5,
                PRTCT=password
```

Where:

- *name* must be the same as the LOCALID parameter of the corresponding DEFINE LINK command. This name is the LU name, the name of the SNA network entity that participates in network sessions with a partner LU. The APPL name (and LOCALID) of the remote Horizon "link" becomes the REMOTEID parameter for defining the partner systems in Model 204. The value of *name* on each APPL definition statement must be unique within the entire network.

Note: Do not confuse the APPL name with the ACBNAME parameter. The ACBNAME parameter refers to the name used when a SNA Communications Server application LU issues a request to open communications with the network. The ACBNAME can be the same as or different from the APPL name, and it does not need to be unique within the network. Its default is the APPL name.

While it can be of some use to code a value for ACBNAME in terminal-to-host systems with multiple copies of the host Online, it is not helpful for host-to-host systems such as Horizon. Moreover, if ACBNAME is coded on the APPL statement, the LOCALID parameter in the DEFINE LINK command must be set to this ACBNAME value, while the REMOTEID parameter in the DEFINE PROCESSGROUP command must still be set according to the APPL name.

The name parameter of the APPL statement for *SNA Communications Server terminal* support must be the same as the VTAMNAME parameter in the User 0 CCAIN of Model 204.

- AUTH (authorize) is required and must specify ACQ (acquire). This specification lets Model 204 acquire sessions with other LUs. In Horizon, Model 204 regularly does this when starting outbound conversations. This parameter setting authorizes the Model 204 SIMLOGON SNA Communications Server request when acquiring sessions.

- Parallel session support, the PARSESS parameter, is required and must be set to YES. This setting causes SNA Communications Server to prepare support—primarily in terms of network addressing tables—for more than one session to take place concurrently between the same pair of LUs.

  All the sessions allowed by the SESSIONS parameter on the Model 204 DEFINE LINK command can take place concurrently with the same remote system, with each bearing a program-to-program conversation.

  Note: If PARSESS parameter defaults to NO, Model 204 fails in its attempt to start a second, simultaneous conversation.

- MODETAB is a required parameter. It ties the APPL definition to the mode definition that specifies the particular set of SNA protocols required for the LU 6.2 session between two LUs.

  MODETAB points SNA Communications Server to the table containing the mode definition. An entry in the table is a set of characteristics, or session parameters, that is applied when a session is established. These parameters are sent from the LU that acquires the session to the partner LU so that both share a common set of rules.

  Generally, unless the environment already contains systems using LU 6.2, the SNA Communications Server systems programmer should construct a new, separate table for LU 6.2 session parameters, with one entry in it. (As described on page 161 and in Appendix E, parallel sessions connections may require two entries.)

  If the entry for the LU 6.2 session parameters is to be placed in a table containing other LU 6.2 entries, the DLOGMOD parameter should be coded (in addition to the MODETAB parameter) to specify which entry is desired.

Note: If MODETAB is not coded at all, SNA Communications Server uses an inappropriate default mode table for the LU 6.2 sessions. Unlike the situation for the mode entry, there is no way for a SNA Communications Server application to supply the name of the proper mode table to SNA Communications Server dynamically upon starting the session; there is accordingly no corresponding DEFINE parameter in Model 204.

Another way to specify the entry is to use the Model 204 DEFINE commands rather than specifying the mode definition in SNA Communications Server. To do so, code the name of the SNA Communications Server mode table entry on the MODENAME parameter of the DEFINE PROCESSGROUP command.

If the mode table specified in MODETAB contains more than one set of session parameters, SNA Communications Server selects the first entry, by default. To specify an entry that is not the first one in the table, you must use either the DLOGMOD parameter in the SNA Communications Server APPL statement or the MODENAME parameter in the Model 204 DEFINE LINK command.

- APPC=NO is required if you are using SNA Communications Server 3.3 or later SNA Communications Server versions. The NO option, which is the default, is required even though Horizon is an LU 6.2 implementation. APPC is not valid for versions of SNA Communications Server earlier than SNA Communications Server 3.3.

- *DLOGMOD* is an optional parameter that specifies the name of a particular entry in the mode table that has the parameters to be used for the session. The value specified for DLOGMOD must match a MODEENT entry name in the specified table.

  DLOGMOD is described further in the description of MODETAB, above.

- VPACING (SNA Communications Server pacing) is an optional parameter. This is the maximum number of messages, or chain elements, received by the LU during a session before an acknowledgment, a pacing response, is sent to the partner LU.

The value of the VPACING parameter must match the value of the SSNDPAC parameter in the partner’s mode table that limits the number of messages sent before a response is received. A value of five (messages) for both is recommended to accommodate average message length.

– SNA Communications Server pacing is also discussed in Appendix C.

– PRTCT (protect password) is an optional parameter that provides a security feature in network access. When the PSWD parameter is coded on the Model 204 DEFINE LINK command, the issuer of the OPEN LINK command is prompted for a password. The password entered must match the value of the PRTCT parameter, or SNA Communications Server rejects the Online request to open communications for the named LU. Chapter 4 fully describes Horizon security features.

Coding the Mode Table Entry for SNA Communications Server Network Definition

As described in “APPL Statements for Network Definition” on page 37, the APPL statement corresponding to a Horizon link points directly to a table, each of whose entries is a set of session characteristics (session parameters). The entire set is actually sent from the LU that acquires the session (the primary LU) to the partner LU (the secondary LU) in the Bind message that establishes the session. The two halves of the session are thereby able to share a common set of rules.

MODETAB, also called logon mode table

The MODETAB table is often called a logon mode table or logmode table and its entries logmodes because, prior to the introduction of program-to-program communications, the table entries referred to the characteristics of a terminal and were set up for the session when the terminal logged on to the network. In a program-to-program session, neither end is a terminal and, strictly speaking, there is no logon being done. However, the notion of a set of session parameters controlling the rules of the session between the two partners still pertains.

Where mode tables reside

Mode tables are assembled and link edited into a load library usually called SYS1.VTAMLIB (in z/OS) or into a file with filetype LOADLIB (in VM). The source code may be entered anywhere, but is usually entered into SYS1.VTAMLST in z/OS or a file with filetype VTAMLST in VM.

Creating a mode table entry

The SNA Communications Server systems programmer must create an entry in the SNA Communications Server mode table to specify the particular set of SNA protocols desired for the session between two Horizon LUs.

This entry is **one** of the following:

- The first or only entry in the table.
- An entry whose LOGMODE parameter value matches the value of the optional DLOGMOD parameter of the APPL statement.

  DLOGMOD parameter of the APPL statement.
- An entry whose LOGMODE parameter value is the same as the MODENAME parameter value of the Model 204 DEFINE PROCESSGROUP command.

Recommended mode table parameter values

Generally you should set the mode table entry with the parameter values shown below. Most of these parameters specify LU type 6.2 variations on basic SNA protocols. Exceptional cases are cited in “Modifications for specific systems and applications” on page 43.

```
tablename MODETAB
entryname MODEENT LOGMODE=entryname
                  TYPE=X’00’,
                  FMPROF=X’13’,
                  TSPROF=X’07’,
                  PRIPROT=X’B0’,
                  SECPROT=X’B0’,
                  COMPROT=X’50B1’,
                  RUSIZES=X’8888’,
                  SSNDPAC=X’05’,
                  PSERVIC=X’060200000000000000102000’
   .        .
   .        .    (Entries for systems other than
   .        .     Model 204 may follow here.)
tablename MODEEND
```

Note: Unlike the case for the APPL name, there is no uniqueness requirement for mode entry names; the same name may appear in more than one table. There is also no need to activate mode tables. The proper table is simply link edited into the library and is accessed from there by as many copies of SNA Communications Server as are designated to use that library.

Modifications for specific systems and applications

The nature of a particular system or application may require modifications to the values of the following parameters of the mode table entry to support Horizon:

- RUSIZES
- SSNDPAC

The COS= parameter in the mode table entry may also need to be modified to meet system and application requirements.

The situations requiring such modifications are discussed in Appendix C.

Modifications for parallel session connections are discussed on page 181. In particular, the PSERVIC value must be different.

Modifications for connections to VAX/VMS systems are discussed on page 169.

Setting the PSERVIC security byte

You may need to review the tenth byte of your PSERVIC setting, which is equivalent to the 23rd byte of the bind between the Horizon partners. PSERVIC describes the layout of the presentation services (PS) portion of the bind.

The tenth byte of PSERVIC governs security options. Three options are relevant:

- X'00' means no security information is accepted by the Horizon primary partner. The secondary partner does not send any user ID or password.
- X'10' means security information is accepted by the Horizon primary partner. The secondary partner sends user ID and password.
- X'12' means security information is accepted by the Horizon primary partner, and the Already-Verified indicator is accepted in lieu of a password. The secondary partner sends user ID but no password.

These settings must correspond to the Horizon DEFINE PROCESSGROUP command LOGIN parameter or Horizon rejects any conversations for the processgroup.

As shown below, the PSERVIC settings must also correspond to the Model 204 SYSOPT parameter X'10' bit setting, which indicates (on or off) whether login is required for access to MODEL 204.

```
X'00' corresponds to LOGIN=NOTRUST, SYSOPT X'10' bit off
X'10' corresponds to LOGIN=NOTRUST, SYSOPT X'10' bit on
X'12' corresponds to LOGIN=TRUST, SYSOPT X'10' bit on
```

Network administration commands

Network administration concerns are those that have to do with the day-to-day operation of the Model 204 network. These operation tasks are governed by the network administration commands. The Rocket Model 204 documentation wiki provides syntax and description for each of the network administration commands:

http://m204wiki.rocketsoftware.com/index.php/Category:Commands

The network administration commands are used to manipulate the different network entities (link, processgroup, and process) once they have been defined. These commands, also called network control commands, can be placed in the User 0 stream or within a Model 204 procedure. They require that the issuer be either User 0, a system administrator, or a system manager. They can also be issued at command level.

The network control commands include:

- OPEN LINK
- CLOSE LINK
- START
- STOP
- MONITOR
- MODIFY

Preparing the ONLINE Configuration

This section describes CCAIN stream parameters the system manager must set appropriately for Horizon to function successfully. The parameters discussed here are the system input (CCAIN) parameters IODEV and NSUBTKS. ONLINE configuration preparation includes allotting Horizon inbound threads (IODEV=27) and adding to the NSUBTSKS parameter value for LU 6.2/SNA Communications Server pseudo subtasks.

IODEV specification for Horizon inbound threads

Processing inbound conversation requests requires that Model 204 have a pool of Horizon threads available for running the server programs. These threads must be defined in the server CCAIN stream as part of the Model 204 ONLINE configuration.

A new IODEV number, 27, defines Horizon inbound conversation threads. The NOTERM parameter is required on the first user parameter line that specifies IODEV=27. NOTERM indicates how many threads are to be allocated for Horizon inbound conversation processing.

To guarantee that there is no shortage of Horizon inbound threads and that all incoming conversation requests can be processed, set the NOTERM parameter to the total of INLIMIT values on all DEFINE PROCESSGROUP commands for the server system.

For more information about IODEV 27, see the Rocket Model 204 documentation wiki: http://m204wiki.rocketsoftware.com/index.php/Defining_the_user_environment_(CCAIN).

Modifying the NSUBTKS parameter

The number of pseudo subtasks that can be used by Model 204 during a run must be increased by 4 per **open** link if Horizon is running in CMS, or by 2 per **open** link if Horizon is running in z/OS.

NSUBTKS is a user zero parameter. It is described in the Rocket Model 204 documentation wiki: http://m204wiki.rocketsoftware.com/index.php/NSUBTKS_parameter

Pseudo subtasks are described in the Rocket Model 204 documentation wiki: http://m204wiki.rocketsoftware.com/index.php/Controlling_system_operations_(CCAIN)

Application Testing with the RPI Subsystem

A server process always invokes an application subsystem. This implies that before a SOUL program can be run as a server process it must first be defined as a Model 204 application subsystem. For running or testing ad hoc procedures, this definition of the procedure as a subsystem may be inconvenient.

The Model 204 Remote Procedure Invocation (RPI) subsystem permits ad hoc procedures that have not been defined as application subsystems to be invoked by server processes. RPI is simply an initiator subsystem that invokes a specified procedure from a specified procedure file.

For more information about developing application subsystems, see the Rocket Model 204 documentation wiki:

http://m204wiki.rocketsoftware.com/index.php/SOUL

RPI subsystem installation

The RPI subsystem is installed automatically and defined to the Application Subsystem facility as part of DICTIONARY installation. It must be installed on the system in which the server program to be tested runs.

System manager responsibilities

Before the RPI subsystem can be used, a system manager has two responsibilities:

1. Ensure that the procedure file containing the procedure to be tested permits the procedure to be invoked.
2. Ensure that the procedure to be tested is defined as a process.

Each of these is described separately below.

Ensuring the procedure file permits procedure inclusion

You cannot include a procedure from the procedure file if the file is private or if it is does not grant sufficient privileges to include a member procedure. You must take the file type and file privileges into account and choose one of two options for opening the procedure file:

1. Open the procedure file automatically when the RPI subsystem is invoked. Do this by adding the procedure file to the RPI subsystem definition using the SUBSYSMGMT facility.
2. Let RPI attempt to open the file. Do this by appropriate coding of the DEFINE PROCESS command SUBSYSPARM parameter, which is discussed below.

The following considerations affect your choice of option:

- RPI can open a public or semipublic procedure file that is not defined to the RPI subsystem as long as the default privileges for the file are sufficient to allow procedure INCLUDEs. For such files, the second option above is simpler.
- RPI cannot open private files and files not given sufficient privileges to allow procedure INCLUDEs. Such files must be added to the RPI subsystem definition to be automatically opened.
- If you rely on the first option above, you must update the RPI subsystem definition whenever you test a procedure from a new procedure file.

Ensuring the procedure to be tested is defined as a process

For each procedure to be tested, supply a DEFINE PROCESS command that includes the following parameters:

```
SUBSYSTEM=RPI
```

```
SUBSYSPARM=’[file] [procedure] [OPEN | NOOPEN]’
```

Where:

- *file* is the Model 204 procedure file containing the procedure.

- *procedure* is the name of the procedure to be invoked.
- OPEN is appropriate if the procedure file is not defined to the RPI subsystem and RPI must try to open it.
- NOOPEN is appropriate if the procedure file is defined to the RPI subsystem and has been automatically opened.

For example, the DEFINE PROCESS command below allows server process ANSWERQ to run the RPI subsystem and invoke the ad hoc procedure MYPROC from TESTFILE, as specified in SUBSYSPARM.

```
DEFINE PROCESS ANSWERQ WITH SCOPE=SYSTEM DATALEN=2048 -
   FROM=BOULDER SUBSYSTEM=RPI                         -
   SUBSYSPARM='TESTFILE MYPROC OPEN'
```

Application developer responsibilities

The application developer testing under RPI must know the following:

- The communication global variable for RPI is COMM. Every procedure being included is responsible for setting this.
- The exit value for COMM is EXIT. The inbound procedure **must** set this when it terminates.

  As always, neglecting to set the communication variable leads to looping or other errors. An inbound process that is looping does not usually call attention to itself until it fills up the CCAAUDIT file.
- You cannot use RPI to test a procedure that expects to receive data in the command line global variable. This variable is reserved for use by RPI.

4

Security

Overview

A distributed transaction consists of at least two parts: the part that requests a remote service (the client), and the part that services the request (the server). This section describes the security concerns of a Horizon network from the point of view of the server and then of the client.

Terminology

Server and client systems are Model 204 ONLINEs that contain application programs that converse with each other. The client program usually communicates directly with a human end user at a terminal. The server program has no access to the terminal. All its communication to the end user must pass through the client program. In addition, the server program is not accessible by terminals attached to the server system, because the service thread is a non-terminal thread.

A system may be a client system or a server system or both.

Contents

Security issues for Horizon are divided into the following three areas:

1. Server system access.

   Managing the use of server system resources by remote and local users. This is discussed in “Managing Access to a Server System” on page 50.

2. Client system use.

   Controlling a client system's local users' access to server systems. This is discussed in "Controlling Users on a Client System" on page 58.

3. Network node name use.

   Protecting network node names from unauthorized use (see page 60).

This chapter provides a description of the features that Model 204 offers to help the system manager control these security issues. A series of examples of different security schemes concludes the chapter.

Managing Access to a Server System

The following issues are of concern to the manager of a server system:

- Controlling which remote client systems may make requests of the server system. This is discussed in "Controlling Remote Client System Access to the Server" on page 51.

- Specifying which individual users from a given client system may use the server system, and which server programs they may use. This is discussed in "Controlling Individual Remote Users" on page 51.

- Protecting the file which contains the service transaction program. This is discussed in "Protecting Procedure Files" on page 57.

Figure 4-1 shows how these issues represent layers of protection for the server system, and how access is controlled. Refer to it during the discussion of access to a server system.

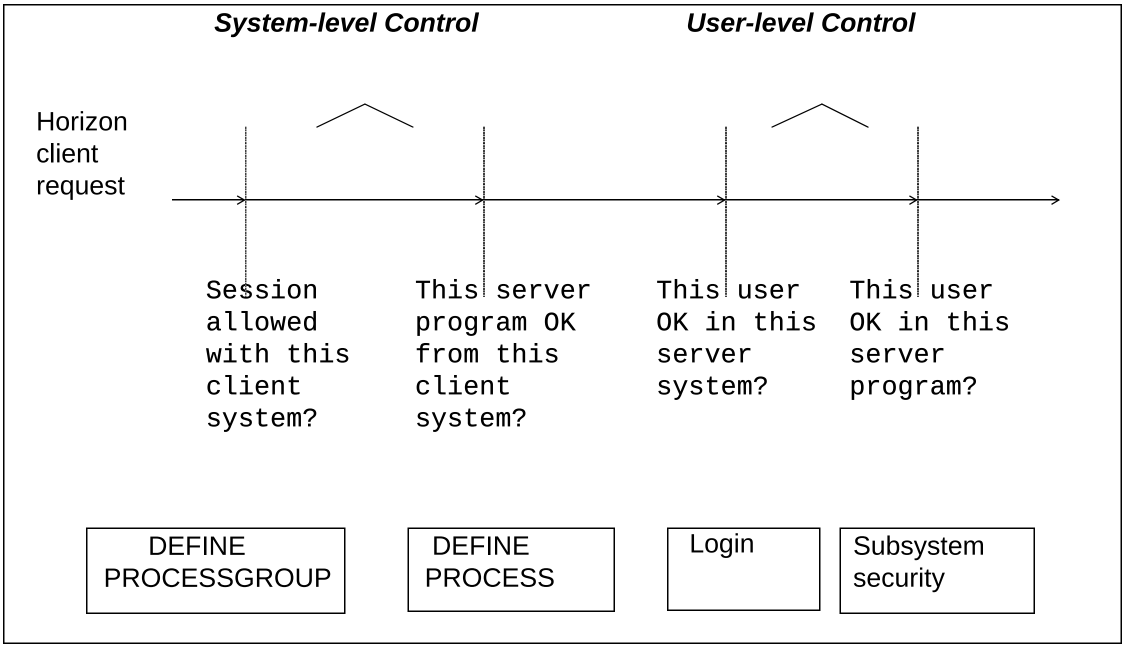

System-level Control
User-level Control
Horizon client request
Session allowed with this client system?
This server program OK from this client system?
This user OK in this server system?
This user OK in this server program?
DEFINE PROCESSGROUP
DEFINE PROCESS
Login
Subsystem security

Figure 4-1 Lines of Defense for a Server System

Controlling Remote Client System Access to the Server

The system manager of a server system controls access to the server system as a whole and to each of its resources individually. This control is exercised through the DEFINE PROCESSGROUP and DEFINE PROCESS commands that precede any use of the Horizon facility.

Controlling access to a server system

For a client system to establish an SNA session with a Model 204 server, the client's network node name must be coded in the REMOTEID parameter of a server's DEFINE PROCESSGROUP command. Since this command requires system manager privileges, a given client system must be specifically authorized by a server's system manager to gain access to the server.

For example, referring to Figure 3-2 on page 25, the following abbreviated DEFINE PROCESSGROUP command allows sessions with the network node M204HQ:

```
DEFINE PROCESSGROUP PGRP1 WITH REMOTEID=M204HQ...
```

Controlling access to a server program

For a given server program (that is, application subsystem or procedure) to be accessible from any client system, the server system manager must include the name of a processgroup that defines that client system in the FROM parameter of the server program's process definition. The system manager is thus required to tell Model 204 explicitly which nodes in the network may access any server program.

For example, referring to Figure 3-2 on page 25, the following abbreviated DEFINE PROCESS command allows access to the server program WSALES from the client system M204HQ (which is defined in processgroup PGRP1):

```
DEFINE PROCESS WSALES FROM=PGRP1...
```

Controlling Individual Remote Users

Model 204 login and Application Subsystem security processing are the principal means by which a system manager controls remote user access to service resources. This section first describes the layers of login processing and possible security options, and then describes Application Subsystem security options.

Login processing

For an end user, the first level of security in Model 204 is login processing. The typical terminal-attached end user presents a password at login time. The system uses the password to verify the user's ID. The ID then can be reliably used to assign operational privileges and to grant access to various resources.

These assignments are made according to rules contained in Model 204 Application Subsystem security and external security packages.

Login processing for a *client user* who logs into a Model 204 system to process part of a distributed transaction differs from that for a typical terminal end-user in the following ways:

- Absence of command level access
- Problematic password use

Access to command level

A successfully logged-in terminal thread accesses a Model 204 system at “command level” (the > prompt) unless the AUTOSYS parameter is specified for the thread. (The AUTOSYS parameter automatically places the user in a subsystem after login to Model 204.) The terminal end user is then permitted certain ad hoc capabilities: to issue certain Model 204 commands, compose and run temporary SOUL Requests, open files, send messages to other users, and so on.

A successfully logged-in Horizon client user, however, is passed directly into a “canned” application: the process to which the user is connecting (which must be predefined by the server’s system manager). Unless that process (server application) is specifically designed to build and execute ad-hoc requests from the client program, the client end user has no ad-hoc capability, because the server thread is never at command level.

To change the server application’s behavior, the end user must be able to log into the server system on a terminal thread and change the server application program.

Password problems

At login time, a terminal end user typically supplies a password at the terminal and is admitted to Model 204 with the appropriate privileges. Supplying a password may not be as easy for a client program, however: when processing a transaction distributed over many server systems, each system involved may require a login from the client. Prompting the end user repeatedly for a password is cumbersome and needlessly exposes the end user to underlying network communication activity. Having the required passwords in a file may be a security exposure.

To resolve this password problem, Horizon login processing introduces options (“trusted” users and “guest” users) for logging in client users that do not present passwords. These options are described in the discussion of Horizon login options that follows.

Login options

In this section, the decisions made during Horizon login processing, as depicted in the branches of the login tree in Figure 4-2, are described in turn, following the figure.

The diagram in Figure 4-2 is a hierarchical representation of the various Horizon security options that affect login for a client user trying to run a server program.

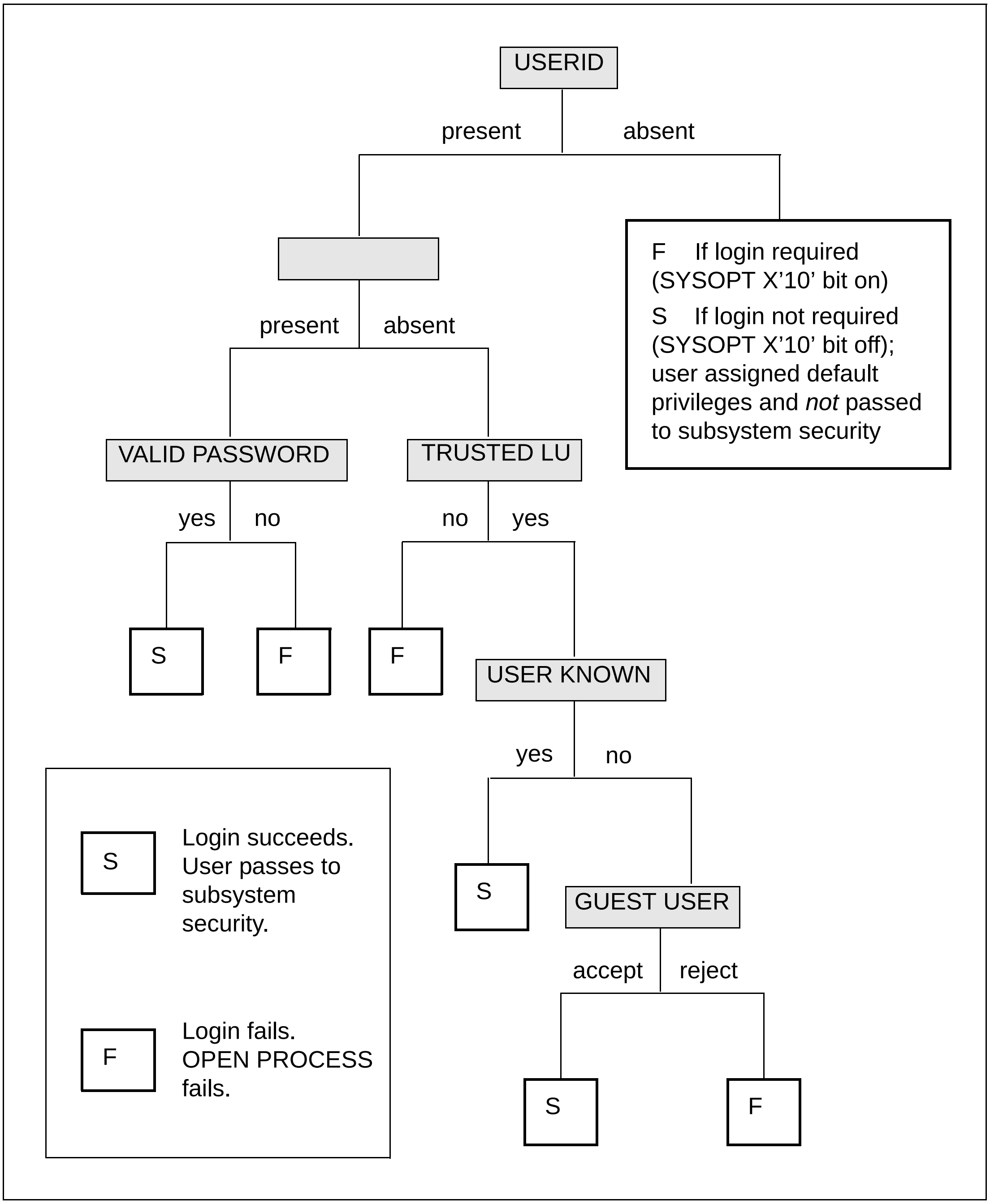

USERID
present
absent
F If login required (SYSOPT X'10' bit on)
S If login not required (SYSOPT X'10' bit off); user assigned default privileges and *not* passed to subsystem security
present
absent
VALID PASSWORD
TRUSTED LU
yes
no
no
yes
S
F
F
USER KNOWN
yes
no
S
GUEST USER
accept
reject
S
F
S
Login succeeds. User passes to subsystem security.
F
Login fails. OPEN PROCESS fails.

Figure 4-2 Login Security Processing

Login required

When the SYSOPT (system options) parameter X'10' bit is on, login is required for any user of the ONLINE. Successful login passes the client user to the server's Application Subsystem security checking, which is reviewed in "Application Subsystem security" on page 56.

Security is described in greater detail in the Rocket Model 204 documentation wiki:
http://m204wiki.rocketsoftware.com/index.php/Storing_security_information_(CCASTAT)

The SYSOPT parameter is described in the Rocket Model 204 documentation wiki: http://m204wiki.rocketsoftware.com/index.php/SYSOPT_parameter

A failure in login processing on the service online system causes the client's OPEN PROCESS request to fail, rejected by the server system.

Login processing first checks for a valid user ID.

USERID

- USERID absent

  If USERID is absent from the incoming OPEN PROCESS request and login is required, login fails.

- USERID present

  If USERID is present, processing proceeds to check for a password.

PASSWORD

- PASSWORD present

  If PASSWORD is present in the incoming request, Model 204 uses it to validate the login.

  – PASSWORD valid

    If the password is valid, login succeeds and the user is passed to Application Subsystem security.

  – PASSWORD invalid

    If the password is not valid, login fails.

- PASSWORD absent

  If the PASSWORD is absent, processing proceeds to check further options.

Trusted LU

If LOGIN=TRUST is specified on the DEFINE PROCESSGROUP command used to define the client to the server, the server system does not require a password to log the user in. The server's system manager uses this option based on the belief that the client node reliably verifies that the requesting user is valid.

Using this option makes it unnecessary to store and pass login passwords throughout the network when processing a distributed transaction. LOGIN=TRUST must be coded on **both** the DEFINE PROCESSGROUP command defining the server to the client and the one defining the client to the server.

The effect of LOGIN in the Horizon login process is as follows:

- LOGIN=NOTRUST

  The client system is not DEFINEd as trusted by the server, and login fails.

- LOGIN=TRUST

  The server system simply accepts the user ID passed in the request from the client system without requiring a password. Processing proceeds to the known user test.

User known

If the client system is trusted, Model 204 attempts to log the user in without a password. A "known" user is one whose USERID value is defined in either the server's password table (CCASTAT) or in the external security package CA-ACF2. (For the security packages Security Server (formerly RACF) and CA-Top Secret, see the Note below.) Login processing of the USERID value on an incoming request proceeds as follows:

- USERID known

  If the requesting USERID value is known, login succeeds and user privileges are as defined in CCASTAT or CA-ACF2. The user is passed to Application Subsystem security.

- USERID unknown

  If the requesting USERID value is not known, login processing proceeds to the guest user test.

Note: USERIDs defined to Security Server or to CA-Top Secret are always considered unknown. If a USERID value is not defined in CCASTAT but is defined in Security Server or CA-Top Secret, the user is considered unknown and login processing proceeds to the guest user test.

Login not required

When the SYSOPT X'10' bit is off, login is not required. An incoming conversation request can proceed even if no user ID is present on the client request.

When login is not required, the outcome of Horizon login processing depends on whether a user ID is present:

- USERID present

  The login is processed just as if login were required:

  – If login requirements are met, Horizon assigns the user the user ID passed in the request. The user is passed to Application Subsystem security checking.

  – If login requirements are not met, login fails and no conversation is allowed.

- USERID not present

  Horizon assigns the user an ID of “NO USERID,” an account of “NO ACCOUNT,” and Superuser privileges.

When login is not required, the resources of the server ONLINE are still protected. With a user ID of NO USERID, the client user cannot be admitted to private application subsystems or to the restricted classes of semipublic subsystems. The user may not open protected files without a valid password and may not issue restricted commands.

A user of a non-Model 204 client system whose LU 6.2 interface does not support security parameters can only connect to a service ONLINE that does not require login.

Application Subsystem security

After login security processing, the following kinds of client users pass to Application Subsystem security processing:

- Those with valid user IDs and passwords
- Those with known user IDs and no passwords but from trusted LUs
- Those without known user IDs but who are “guests” from trusted LUs
- Those with user IDs but for whom login processing was not required

Client users must pass through Application Subsystem security before they can execute any service transaction. Based on the user ID and the type of the subsystem, Application Subsystem security decides:

- Who is allowed to enter a given subsystem
- What privileges are granted upon entry
- What kind of access to subsystem-owned files is granted

This section contains a brief summary of how the subsystem handles these three security concerns. For a more thorough description of application subsystems, see the Rocket Model 204 documentation wiki:
http://m204wiki.rocketsoftware.com/index.php/System_requirements_for_application_subsystems

Controlling who is allowed in

Subsystems can be public, semipublic, or private. They differ in how they treat attempts to enter the subsystem.

- A public subsystem permits all users to enter and gives all users the same level of privileges.
- A semipublic subsystem permits access to all users. A user defined to the subsystem gets the privileges associated with that user class (SCLASS). Undefined users get the privileges associated with the default SCLASS.
- A private subsystem admits only users who are defined to it.

How privileges are assigned

Individual privileges held at entry to the subsystem can be overridden by the user's SCLASS assignment. The system manager may assign different privileges for each SCLASS.

The privileges set in the SCLASS definition are equivalent to the user privileges ordinarily granted upon successful login to Model 204: the power to create files, issue certain privileged commands, change passwords, or access secured records.

For more information about these privileges, refer to the Rocket Model 204 documentation wiki:
http://m204wiki.rocketsoftware.com/index.php/System_requirements_for_application_subsystems

The SCLASS may also define the account name the user is assigned while in the subsystem, and may also define the record security key for the user.

This assignment of privileges, account name, and record security key lasts only while the user is in the subsystem in question. If the user transfers to another subsystem, the new subsystem determines privileges, account, and record security.

How access to subsystem files is assigned

SCLASS definition also controls what files a user may open and what file privileges are granted per file.

Protecting Procedure Files

The service program resides in a procedure file in the server ONLINE. This file can be protected from access by end users on the server system by making the file semipublic or private, that is, by requiring a password. The file is opened automatically upon entry to the server subsystem, making a password unnecessary for client system users.

Controlling Users on a Client System

This section discusses the following issues of concern to the manager of a client system:

- Specifying which remote server systems may be accessed from the local system
- Specifying which remote server definitions are available for ad hoc use and which are to be restricted to application subsystems
- Controlling the ways in which individual users are allowed to identify themselves to a server system

Defining remote server systems

Using the DEFINE PROCESSGROUP and DEFINE PROCESS commands, the system manager of the client system defines the remote server systems and programs with which the client can communicate.

- DEFINE PROCESSGROUP

  Just as the server system requires that each remote client system be defined to it in a DEFINE PROCESSGROUP command, the client system requires that any server system to which it passes a request be defined the same way: through the REMOTEID parameter.

- DEFINE PROCESS

  Each remote service must be defined in a DEFINE PROCESS command. The DESTINATION parameter specifies to which service systems an OPEN PROCESS request for this transaction may be directed.

Restricting access to remote server systems

A distributed application can be designed in such a way that the server side of the application is responsible for all checking to make sure that its client is authorized to issue incoming requests. However, having the server prevent malicious or incorrect client programs from compromising it may be inconvenient. It is often easier for the server program to assume that it is conversing with a correctly debugged, trustworthy client. This assumption implies two requirements:

1. The code of the client program is protected from unauthorized modification.
2. An unauthorized program is prevented from “posing” as the client.

Restricting client programs to application subsystems achieves this protection.

An application subsystem provides a secure environment in which to package and run the client program, thus providing the first protection above. Access to a client program can be limited to predefined users. The behavior of the client program can be further restricted by the definition of the SCLASS to which the end user is assigned. Unauthorized end users are prevented from changing either the client or server.

The second protection listed above is provided by blocking the access of non-subsystem clients. In SOUL, a program establishes itself as a client by issuing the OPEN PROCESS statement to initiate a conversation with a server. To block an OPEN PROCESS statement issued by a client running outside the application subsystem environment, the client system manager can use the RESTRICT parameter of the DEFINE PROCESS command, as follows:

```
RESTRICT=APSY
```

When RESTRICT=APSY is coded, end users outside the subsystem cannot run their own copy of a client program. They must use the subsystem's copy.

Controlling user identification to a remote server system

A client program can pass to a server system a user ID, password, and account (or profile) other than the current value of these items by supplying new values in the USERID, PASSWORD, and ACCOUNT (or PROFILE) parameters of the OPEN PROCESS statement.

To control any such client program changing of USERID and ACCOUNT (or PROFILE), the system manager of the client system can choose one of the following three options for the DEFINE PROCESS command UIDSOURCE, ACCTSOURCE, or PROFSOURCE parameters:

- CURRENT
- OPEN
- NONE

Using the CURRENT option with UIDSOURCE and ACCTSOURCE (or PROFSOURCE)

If CURRENT is specified, the client program cannot change the user ID or account number (or profile) that the server sees by coding them on the OPEN PROCESS statement. If these parameters appear on the OPEN PROCESS statement, the OPEN fails. The client's OPEN PROCESS passes to the server only the user ID and account with which the user is currently logged in.

This option is designed to be used in combination with the LOGIN=TRUST option of DEFINE PROCESSGROUP. Even if the PROCESS definition specifies UIDSOURCE=CURRENT, the OPEN PROCESS statement may contain a PASSWORD parameter (unless the partner is a non-Model 204 program); but if the conversation is initiated with a TRUSTed server, the password is not needed.

Using the OPEN option with UIDSOURCE and ACCTSOURCE (or PROFSOURCE)

OPEN allows the client program to change the user ID or account that the server sees. The client may use values other than those currently logged in to the server by specifying them on the OPEN PROCESS statement.

If the client specifies the USERID parameter on OPEN PROCESS, the PASSWORD parameter must also be supplied. ACCOUNT (or PROFILE) may be changed without supplying PASSWORD. If a program omits the USERID or ACCOUNT (or PROFILE) parameter, the user’s current user ID or account is passed.

Using the NONE option with UIDSOURCE and ACCTSOURCE (or PROFSOURCE)

The NONE option suppresses all security information in the OPEN request. It is useful when the server system does not require login. It may be required in future Horizon implementations if the server system is a non-Model 204 system that does not accept security parameters from clients. Such a system may reject an OPEN request that includes security parameters.

Table 4-1 displays the effects of the UIDSOURCE and ACCTSOURCE options, depending on what the OPEN PROCESS contains. The results in the table are the same if PROFILE and PROFSOURCE are substituted for ACCOUNT and ACCTSOURCE, since they are synonyms.

Table 4-1. Selection of User ID and Account Number

		Value of UIDSOURCE and ACCTSOURCE:		
		CURRENT	OPEN	NONE
USERID or ACCOUNT coded on OPEN PROCESS statement:	Yes	OPEN fails	Taken from OPEN PROCESS	OPEN fails
	No	Current	Current	None

Protecting Network Node Names

The system manager authorizes each network node that a given Model 204 system may communicate with by specifying its name in the REMOTEID parameter of a DEFINE PROCESSGROUP command. Certain options of the DEFINE PROCESSGROUP command (LOGIN=TRUST and

GUESTUSER=ACCEPT) grant special status to the node named in the REMOTEID parameter. To confidently designate a partner system as trusted, a system manager must be assured of the following:

- The only user of that node name is the system that the user is authorizing.
- Individuals cannot bring up their own unauthorized system that identifies itself to the network using an authorized node name.

Using SNA Communications Server password protection

SNA Communications Server (formerly VTAM)'s APPL statement, which defines a network node name, provides an option to restrict use of that name. The PRTCT parameter, when coded, contains a password that must be supplied when the SNA Communications Server ACB is opened. This feature can be used to insure that certain network node names are used only by authorized systems.

If PRTCT is coded on the APPL definition, the PSWD parameter must be included in the DEFINE LINK command. PSWD causes the OPEN LINK command to prompt for a password. If the password given does not match the one which appears in the APPL PRTCT parameter, the OPEN LINK command fails.

For more information

Coding the SNA Communications Server APPL definition is described further in "Defining the Network to SNA Communications Server" on page 36. The DEFINE LINK command is described on page 26.

Security Design Examples

Horizon was designed to place responsibility for understanding network topology and security in the hands of the system manager and to shield the application programmer from these considerations. This section contains a series of security design examples that increase in complexity and show some of the range of possible security options. Starting with a very simple design, which uses a minimum of features, new options are added as they are needed to satisfy the given security needs of each example.

Six security design examples are provided. In each example, security features that are new or changed from the previous example are noted. The following table identifies the example number and its features:

Example	New or highlighted feature
1.	No server or client security: default security parameters.
2.	Server has login security protection.

Example	New or highlighted feature
3.	Server trusts client; server-client link is SNA Communications Server-password protected; client system is login protected.
4.	Server trusts client; client programs can only be application subsystems.
5.	Two trusted client processes: one restricted to application subsystems and one, unrestricted, for ad hoc requests. Server can differentiate between the two processes.
6.	Two trusted client processes; application subsystem security in effect for server.

Example 1: Login Not Required on Server

By default, no security options apply to a Horizon conversation. The server system must be configured with the SYSOPT=X’10’ bit off (login not required) to permit a client (who is not passing a user ID) to OPEN a PROCESS for which no security attributes have been specified.

This section contains displays of sample client and server coding for this example, followed by notation of the security aspects featured. The client and server DEFINE commands and SOUL requests use the default security parameters, which are coded explicitly and accompanied by comments. Most non-security related parameters are omitted.

Coding for client

Client Network Definition	Comments
`DEFINE LINK LINKCLI WITH      -`	
`  LOCALID=M204HQ              -`	
`  NOPSWD`	*No LINK password prompt*
`DEFINE PROCESSGROUP PGCLI WITH-`	
`  LINK=LINKCLI                -`	
`  REMOTEID=M204BO             -`	
`  LOGIN=NOTRUST               -`	*Partner LU not trusted*
`  GUESTUSER=REJECT`	*Login of unknown user fails*
`DEFINE PROCESS WKSALES WITH   -`	
`  DESTINATION=PGCLI           -`	*Connection only allowed to M204BO*
`  PARTNER=WSALES              -`	
`  UIDSOURCE=NONE              -`	*Send no user ID in OPEN request*
`  ACCTSOURCE=NONE             -`	*Send no ACCOUNT in OPEN request*
`  RESTRICT=NONE`	*Ad hoc request may OPEN*

Client Conversation	Comments
`BEGIN`	
`  OPEN PROCESS WKSALES`	*No USERID, PASSWORD, or ACCOUNT*
`  .`	
`  .`	
`  converse`	
`  .`	
`  .`	
`  CLOSE PROCESS WKSALES`	
`END`	

Coding for server

Server Network Definition	Comments
`SYSOPT=X’nn’ (X’10’ bit off)`	*Login not required*

`DEFINE LINK LINKSRV WITH       -`
`  LOCALID=M204BO               -`
`  NOPSWD`
 	
`DEFINE PROCESSGROUP SALES WITH -`
`  LINK=LINKSERV                -`
`  REMOTEID=M204HQ              -`
`  LOGIN=NOTRUST`
`  GUESTUSER=REJECT`
 	
`DEFINE PROCESS WSALES WITH     -`
`   FROM=SALES                  -`
`   SUBSYSTEM=WKS`
	
	No LINK password prompt
	
	Partner LU not trusted
	Logon of unknown user fails
	
	Connection only allowed from
	M204HQ
	Invoke WKS subsystem

Server Conversation

```
BEGIN
  OPEN PROCESS internalname ACCEPT
   .
   .
  converse
   .
   .
  CLOSE PROCESS internalname
END
```

Notes

1. The server system is not necessarily unsecured:
   - The system can be configured solely for use by client users, so that the files are protected by Application Subsystem security.
   - The system manager can password protect the ONLINE’s files.
   - The ONLINE system may be configured with no terminal threads. The system manager may define only server threads (IODEV=27).
2. The only source authorized as a client user of the WKS subsystem is Headquarters (M204HQ).
3. The WKS subsystem must be either semipublic or public: the client user is sending no security parameters and therefore enters the subsystem with a user ID of NO USERID and an ACCOUNT value of NO ACCOUNT. A private subsystem requires a user ID other than NO USERID.

Example 2: Login Required on Server

In this example, the system manager of the server system from the previous example adds an end user application to that system, as well as some public files which end users can open at will. To control which users can enter the server ONLINE, the server system manager turns on X’10’ of the SYSOPT parameter, causing the system to require login. The system manager of the server system informs the system manager of the client system at Headquarters that client Horizon requests fail without a user ID. User IDs presented are validated if they are defined in the server’s CCASTAT file or in an external security database such as ACF2.

In this and following examples, the defaulted values are removed. Changes to the example configurations are indicated by two plus signs (++) to the left of the line.

Coding for client

	Client Network Definition	Comments
	`DEFINE LINK LINK1 WITH           -`	
	`  LOCALID=M204HQ`	
	`DEFINE PROCESSGROUP PGRP1 WITH -`	
	`  LINK=LINK1                     -`	
	`  REMOTEID=M204BO`	
	`DEFINE PROCESS WKSALES WITH      -`	
	`   DESTINATION=PGRP1             -`	*Connection only allowed to M204BO*
	`   PARTNER=WSALES                -`	
++	`   UIDSOURCE=OPEN`	*Allow OPEN PROCESS to code USERID*

	Client Conversation	Comments
	`BEGIN`	
	`    %UID=$READ('ENTER USERID   -`	
	`    %PSWD=$READ('ENTER`	
	`PASSWORD')`	
	`    OPEN PROCESS WKSALES WITH  -`	
++	`    USERID %UID                -`	*User ID specified for server*
++	`    PASSWORD %PSWD`	*Login password*
	`    .`	
	`    .`	
	`    converse`	
	`    .`	
	`    .`	
	`    CLOSE PROCESS WKSALES`	
	`END`	

Coding for server

	Server Network Definition	Comments
++	SYSOPT=X’nn’ (X’10’ bit on)	*Login required*
	DEFINE LINK LINKSRV WITH - LOCALID=M204BO -	

	DEFINE PROCESSGROUP SALES WITH -
	LINK=LINKSRV -
	REMOTEID=M204HQ -	

	DEFINE PROCESS WSALES WITH -	
	FROM=SALES -	*Connection only allowed from M204HQ*
	SUBSYSTEM=WKS	*Invoke WKS subsystem*

Server Conversation
BEGIN OPEN PROCESS internalname ACCEPT . . converse . . CLOSE PROCESS internalname END

Notes

1. The only parameters changed in this example are SYSOPT on the server side and UIDSOURCE on the client side.
2. The USERID and PASSWORD parameters are added to OPEN PROCESS on the client side to allow the Horizon request to successfully log in to the server system.
3. Each user of the client request is prompted for a user ID and password to be passed to the server system.
4. The server’s system manager must now define each authorized client user from M204HQ, as well as each end user who uses his ONLINE, in either the CCASTAT file or an external security database, for example, ACF2.

Example 3: Trusting the Client

Each time that an end user on the client ONLINE wants to run the Horizon request, the user must enter a user ID and password to access the server node. To relieve this burden, the application is changed so that the client system is trusted. A client user is prompted for a password only upon logging in to the client system.

Coding for client

	Client Network Definition	Comments
	`DEFINE LINK LINKCLI WITH        -`	
	`  LOCALID=M204HQ                -`	
++	`  PSWD`	*Prompt for password at OPEN* LINK
	`DEFINE PROCESSGROUP PGCLI WITH  -`	
	`  LINK=LINKCLI                  -`	
	`  REMOTEID=M204BO               -`	*Login won't require password*
++	`  LOGIN=TRUST`	
	`DEFINE PROCESS WKSALES WITH     -`	
	`  DESTINATION=PGCLI             -`	*Connection only allowed to M204BO*
	`  PARTNER=WSALES                -`	
++	`  UIDSOURCE=CURRENT             -`	*OPEN PROCESS passes current user ID*

	Client Conversation	Comments
	`BEGIN`	
++	`    OPEN PROCESS WKSALES`	*No user ID or password need be coded anymore.*
	`    .`	
	`    .`	
	`    converse`	
	`    .`	
	`    .`	
	`    CLOSE PROCESS WKSALES`	
	`END`	

Coding for server

```
      Server Network Definition         Comments

      DEFINE LINK LINKSRV WITH       -
         LOCALID=M204BO              -

      DEFINE PROCESSGROUP SALES WITH -
         LINK=LINKSRV                -
         REMOTEID=M204HQ             -
 ++      LOGIN=TRUST

      DEFINE PROCESS WSALES WITH     -
         FROM=SALES                  -  Connection only allowed from M204HQ
         SUBSYSTEM=WKS                  Invoke WKS subsystem
```

```
Server Conversation

BEGIN
    OPEN PROCESS internalname ACCEPT
    .
    .
    converse
    .
    .
    CLOSE PROCESS internalname
END
```

Notes

1. Both the client and server processgroup definitions are changed from LOGIN=NOTRUST to LOGIN=TRUST. This attribute must be the same on both partners' processgroup definitions for Horizon to work.

2. The PSWD parameter is added to the LINK definition. Since these partners are to TRUST each other, they must be protected from a situation in which an impostor opens a link using one of their LOCALID names. Specifying PSWD causes the password prompt to be displayed at OPEN LINK time; but password verification is initiated by adding the PRTCT parameter to the SNA Communications Server APPL definition for each LOCALID. Note that this (SNA Communications Server) password is completely unrelated to any login password and has nothing to do with the CCASTAT file.

3. The client process definition has UIDSOURCE=CURRENT, prohibiting the SOUL program from specifying its own user ID on OPEN PROCESS.

4. The client system passes (without a password) the end user's currently logged in user ID to the server system. Since the client system is another Model 204 system, the server's system manager can trust that the client system verified the end user's password before allowing that user to operate with that user ID. This is true even if login is not required on the client

system: if the end user does not log in or if their login fails, they proceed with a user ID of NO USERID.

5. The end user no longer has to pass a password for a conversation with the M204BO system, and may not even be aware that the application does any underlying communication.

6. Any number of trusted systems can be defined this way.

Example 4: Protecting the Client Process Definition

The sample application is revised so that the service program now performs some updates. (Note that only one node is updated by a given transaction. As discussed in “Remote Updating Example” on page 134, a two-node update can cause problems at recovery time.) An incorrectly written client request that OPENs the PROCESS may cause data corruption on the server system. To prevent ad hoc client requests from using the process definition, the client system manager makes the following adjustment.

Coding for client

Client Network Definition	Comments
`DEFINE LINK LINKCLI WITH       -`	
`  LOCALID=M204HQ               -`	
`DEFINE PROCESSGROUP PGCLI WITH -`	
`  LINK=LINKCLI                 -`	
`  REMOTEID=M204BO              -`	
`  LOGIN=TRUST`	*Login won’t require password*
`DEFINE PROCESS WKSALES WITH    -`	
`  DESTINATION=PGCLI            -`	*Connection only allowed to M204BO*
`  PARTNER=WSALES               -`	
`  UIDSOURCE=CURRENT            -`	*OPEN PROCESS passes current user ID*
++ `  RESTRICT=ASPY`	*OPEN only under application subsystem*

Client Conversation	Comments
`BEGIN`	
`    OPEN PROCESS WKSALES`	*No user ID or password need be coded anymore.*
`    .`	
`    .`	
`    converse`	
`    .`	
`    .`	
`    CLOSE PROCESS WKSALES`	
`END`	

Coding for server

Server Network Definition	Comments
`SYSOPT=X’nn’ (X’10’ bit on)`	*Login required*

`DEFINE LINK LINKSRV WITH      -`
`    LOCALID=M204BO            -`
 	
`DEFINE PROCESSGROUP SALES WITH -`
`    LINK=LINKSRV              -`
`    REMOTEID=M204HQ           -`
`    LOGIN TRUST`
 	
`DEFINE PROCESS WSALES WITH -`
`    FROM=SALES                -`
`    SUBSYSTEM=WKS`
	Connection only allowed from M204HQ
	Invoke WKS subsystem

Server Conversation

`BEGIN`
`    OPEN PROCESS internalname ACCEPT`
`    .`
`    .`
`    converse`
`    .`
`    .`
`    CLOSE PROCESS internalname`
`END`

Notes There is only one difference: RESTRICT=APSY has been added to the process definition on the client side.

Example 5: Two Different Process Definitions: Inquiry and Update

Some users on the client system require ad hoc access to the process definition to do inquiries. The solution is to define two separate processes: one for inquiry-only work, accessible to ad hoc requests, and another for updates, which may only be used by a canned application subsystem.

Coding for client

	Client Network Definition	Comments
	DEFINE LINK LINKCLI WITH -	
	LOCALID=M204HQ	
	DEFINE PROCESSGROUP PGCLI WITH -	
	LINK=LINKCLI -	
	REMOTEID=M204BO -	
	LOGIN=TRUST	*Login won’t require password*
++	DEFINE PROCESS WKSALESU WITH -	
	DESTINATION=PGCLI -	*Connection only allowed to M204BO*
++	PARTNER=WSALESU -	
	UIDSOURCE=CURRENT -	*OPEN PROCESS passes current user ID*
	RESTRICT=APSY	*OPEN only under application subsystem*
	DEFINE PROCESS WKSALES WITH -	
	DESTINATION=PGCLI	*Connection only allowed to M204BO*
	PARTNER=WSALES -	
	UIDSOURCE=CURRENT	*OPEN PROCESS passes current user ID*
++	RESTRICT=NONE	*May be opened by ad hoc request*

	Client Conversation	**Comments**
	`BEGIN`	*Ad hoc request*
	`    OPEN PROCESS WKSALES`	*User inquiry-only process*
	`    .`	
	`    .`	
++	`    Make inquiry`	
	`    .`	
	`    .`	
	`    CLOSE PROCESS WKSALES`	
	`END`	

	Client Conversation	**Comments**
	`BEGIN`	*Application subsystem request*
	`    OPEN PROCESS WKSALESU`	
	`    .`	
	`    .`	
++	`    Make inquiry and update`	*User query and update process*
	`    .`	
	`    .`	
	`    CLOSE PROCESS WKSALES`	
	`END`	

Coding for server

Server Network Definition	Comments
`SYSOPT=X'nn' (X'10' bit on)`	*Login required*
`DEFINE LINK LINKSRV WITH -`	
`LOCALID=M204BO -`	
`DEFINE PROCESSGROUP SALES WITH -`	
`LINK=LINKSRV -`	
`REMOTEID=M204HQ -`	
`LOGIN=TRUST`	
`DEFINE PROCESS WSALES WITH -`	*Inquiry service program*
`FROM=SALES -`	*Connection only allowed from M204HQ*
`SUBSYSTEM=WKS -`	*Invoke WKS subsystem*
`SUBSYSPARM='INQUIRY'`	*Command-line parameter*
`DEFINE PROCESS WKSALESU WITH -`	*Updating service program*
`FROM=SALES -`	*Connection only allowed from M204HQ*
`SUBSYSTEM=WKS -`	*Invoke WKS subsystem*
`SUBSYSPARM='UPDATE'`	*Command-line parameter*

Server Conversation	Comments
`BEGIN`	
`OPEN PROCESS internalname ACCEPT`	
`.`	
`.`	
`IF request to update received`	
`IF %CMDLINE = 'UPDATE'`	
`CLOSE PROCESS internalname -`	*Reject request*
`ERROR`	
`END IF`	
`END IF`	
`.`	
`.`	
`CLOSE PROCESS internalname`	
`END`	

Notes

1. There is a new process definition on the client side. The old name, WKSALES, was used for the inquiry-only process definition. The new definition, exclusive to the application subsystem, uses the name WKSALESU.

2. On the server side, there are two process definitions, both invoking the same subsystem. However, the SUBSYSPARM parameter is added, which causes the quoted value to appear in the command-line variable when the subsystem is entered. This allows the subsystem to differentiate

between a client user with only inquiry privileges and one with update privileges.

Note: The same end user can enter the service subsystem with different privileges:

- Writing one’s own client request and entering with INQUIRY
- Using the application subsystem on the client side to enter with UPDATE
- Or, depending upon the SCLASS assigned, this end user may not have update privileges for any subsystem file.

Example 6: Accepting Guest Users

As more and more users on the client system want to use this application, the server system manager must define each client user in the server CCASTAT file or in an external security database. Since these users can only run in one application subsystem of the server system, the following revision places the responsibility for the subsystem’s security in the hands of its administrator, and it takes the server’s system manager out of the loop.

Coding for the client

Client Network Definition	Comments
`DEFINE LINK LINKCLI WITH -`	
`  LOCALID=M204HQ`	
`DEFINE PROCESSGROUP PGCLI WITH -`	
`  LINK=LINKCLI -`	
`  REMOTEID=M204BO -`	
`  LOGIN=TRUST`	*Login won’t require password*
`DEFINE PROCESS WKSALESU WITH  -`	
`  DESTINATION=PGCLI -`	*Connection only allowed to M204BO*
`  PARTNER=WSALESU -`	*OPEN PROCESS passes to current user ID*
`  UIDSOURCE=CURRENT -`	*May only be OPENed under*
`  RESTRICT=APSY`	*application subsystem*
`DEFINE PROCESS WKSALES WITH   -`	*Connection only allowed to M204BO*
`  DESTINATION=PGCLI           -`	
`  PARTNER=WSALES              -`	
`  UIDSOURCE=CURRENT-`	*OPEN PROCESS passes to current user ID*
`  RESTRICT=NONE`	*May be opened by ad hoc request*

Client Conversation	Comments

BEGIN
   OPEN PROCESS WKSALES
     .
     .
   Do inquiry
     .
     .
   CLOSE PROCESS WKSALES
END
	Ad hoc request
	Uses inquiry-only process

Client Conversation	Comments

BEGIN
   OPEN PROCESS WKSALESU
     .
     .
   Do inquiry and update
     .
     .
   CLOSE PROCESS WKSALESU
END
	Application subsystem request
	Uses update process

Coding for the server

	Server Network Definition	Comments
	`SYSOPT=X’nn’ (X’10’ bit on)`	*Login required*
	`DEFINE LINK LINKSRV WITH       -`	
	`   LOCALID=M204BO              -`	
	`DEFINE PROCESSGROUP SALES WITH -`	
	`   LINK=LINKSRV                -`	
	`   REMOTEID=M204HQ             -`	
	`   LOGIN=TRUST -`	
++	`   GUESTUSER=ACCEPT`	*Login of unknown user succeeds*
	`DEFINE PROCESS WSALES WITH     -`	*Inquiry service program*
	`  FROM=SALES                   -`	*Connection only allowed from M204HQ*
	`  SUBSYSTEM=WKS                -`	*Invoke WKS subsystem*
	`  SUBSYSPARM=’INQUIRY’`	*Command-line parameter*
	`DEFINE PROCESS WSALESU WITH    -`	*Updating service program*
	`  FROM=SALES                   -`	*Connection only allowed from M204HQ*
	`  SUBSYSTEM=WKS                -`	*Invoke WKS subsystem*
	`  SUBSYSPARM=’UPDATE’`	*Command-line parameter*

Server Conversation	Comments
` OPEN PROCESS internalname ACCEPT`	
`   .`	
`   .`	
` IF request to update received`	
`   IF %CMDLINE ¬= ’UPDATE’`	
`     CLOSE PROCESS internalname -`	*Reject request*
`      ERROR`	
`   END IF`	
` END IF`	
`   .`	
`   .`	
` CLOSE PROCESS internalname`	
`END`	

Notes

1. The only parameter change is the addition of the GUESTUSER=ACCEPT parameter on the processgroup definition on the server side. Client users from this particular processgroup no longer must be defined in CCASTAT.

2. If the service subsystem is private, the subsystem administrator must define each prospective client user to the subsystem. If it is semipublic, only those client users that are in SCLASSes other than the default must be defined.

5

Horizon Conversation Interface

Overview

This chapter is intended for programmers who use the Horizon conversation interface, the statements a SOUL request uses to converse with a partner program. The SOUL conversation statements that comprise the conversation interface are discussed in detail individually and shown in sample programs in this chapter.

Horizon partners

Horizon partners have a ***peer to peer*** relationship. One partner controls the direction of data flow on the path at any given time, but either partner may assume control at different points in the conversation. Which partner is in control at a given point is decided by the application designer and is based on Horizon’s implementation of the LU 6.2 conversation protocols. The rules that govern Horizon conversations are described and their use is illustrated in this chapter.

Conversation Rules and States

Imagine a debate in which there are no rules of discourse: each side talks out of turn, arguing its point at the same time as the other. It would be difficult, if not impossible, to follow each train of thought and make sense of it. A formal set of rules prevents this, however, by requiring each side to talk and listen in turn.

Two communicating programs also need to obey rules. An orderly conversation implies, for example, that a program that sends data can expect the other to receive it:

```
Partner A                 Partner B

SEND      ---------->     RECEIVE
SEND      ---------->     RECEIVE
```

An orderly conversation also implies that both programs do not wait to receive data at the same time, each expecting the other to send it:

```
Partner A               Partner B

SEND      ---------->   RECEIVE
RECEIVE                 RECEIVE
```

Rules impose order on a conversation by specifying what actions either partner may take at any given point. Rules also allow the LU that is host to a conversation program to detect errors and to report them in an understandable way to the programmer. Without these rules, programmers would be forced to detect errors on their own by observing lost data or deadlock conditions.

Conversation states

LU 6.2 conversation rules rely on the concept of a conversation state. At the completion of each SOUL communication statement, the conversation is said to be in one of a number of ***states***. Each state is associated with a set of ***verbs*** (communication statements) the application program is allowed to issue when in that state, and a set of verbs that are prohibited when in that state. A set of rules also dictates how a conversation partner may change from one state to another upon the completion of a verb (in Horizon, a SOUL statement).

Violations of these state rules are detected by the LU at which a program is running. When a violation occurs, the LU returns a ***state check*** indicator to the program, through the $STATUS and $STATUSD functions. State checks are described in more detail in “Enforcing direction of data flow” on page 82.

Horizon conversation states

A Horizon conversation partner can be in one of five states:

State	Description
Reset	Exists between conversations. No Horizon resources have been acquired; the process is not open.

State	Description
Send	The state in which a partner is allowed to send data.
Receive	The state in which a partner must be to receive information.
Confirm	An intermediate state in which the receiving side of the conversation is expected to send a confirmation or an error response to its partner.
Close	The state in which the conversation is over (the partner has closed its process), but local resources are still allocated.

Horizon state rules

The transition rules that specify how a partner changes from one state to another are described in the examples in this section. The descriptions of the individual SOUL conversation statements in Chapter 6 contain tables that list the state changes possible for each statement. Table A-1 in Appendix A lists the Horizon statements that can be issued in each state.

Horizon state rules also help to address the following issues, each of which is discussed in detail in this section:

1. Enforcing the direction in which data may be sent at a given point in the conversation
2. Providing ways for the receiving partner to interrupt the sender and request the right to send data
3. Providing a way for the sending partner to request acknowledgment that sent data has been received and accepted, and for the receiving partner to supply that acknowledgment

Horizon Conversation Data Flow

Data flow rules

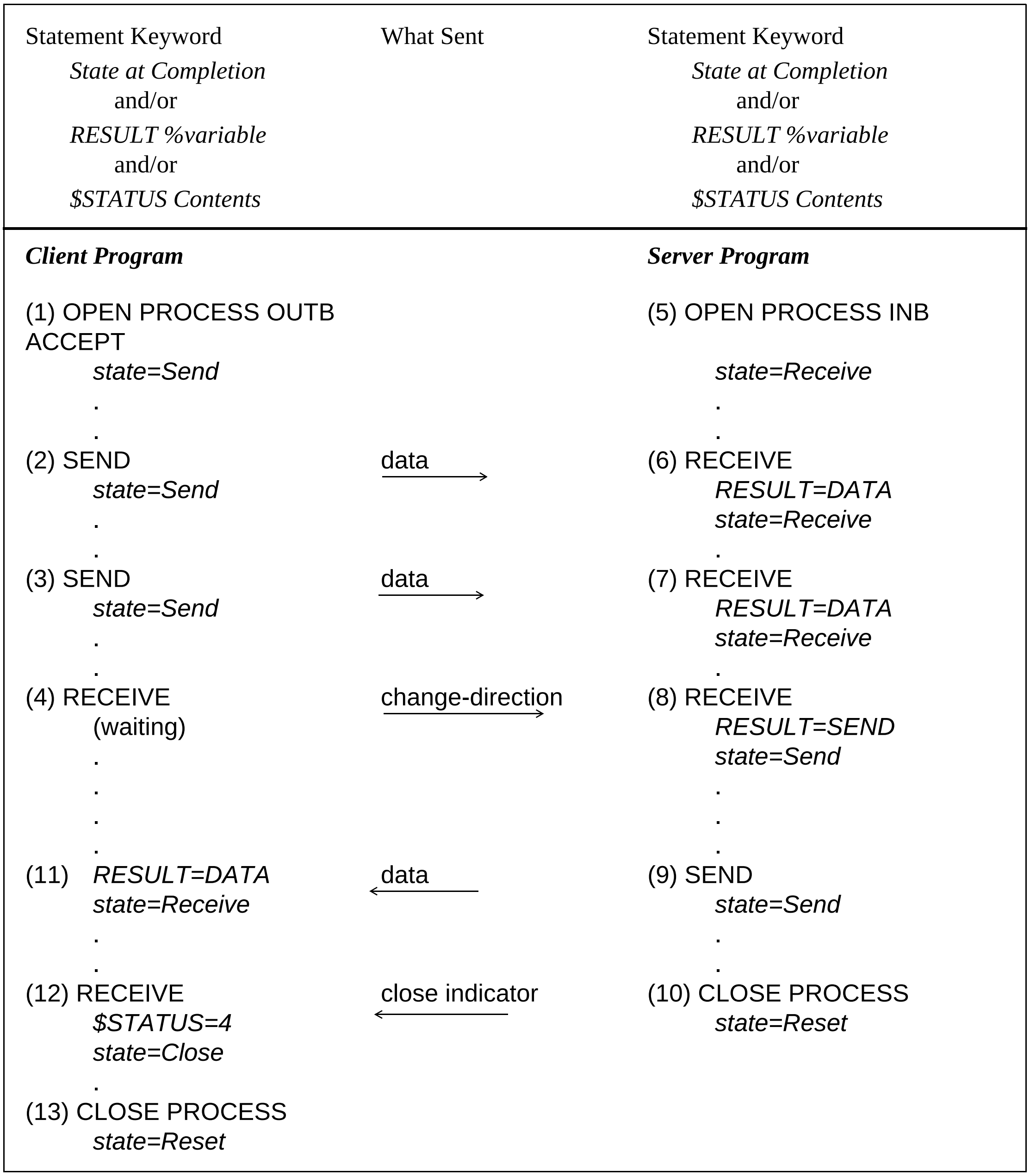
The following rules govern Horizon conversation data flow:

- When a conversation begins, the client side is in Send state and the server side is in Receive state.
- At any given time during a conversation, one and only one partner is allowed to send data. The other must receive it.
- The direction of data transfer remains the same until the side that is sending indicates that it is finished. A program indicates that it is finished sending data by issuing one of the following statements:
  - RECEIVE, which allows its partner to send data.
  - CONFIRM, to which its partner must respond positively or negatively.
  - CLOSE PROCESS, which terminates the conversation.

Each conversation partner runs under the supervision of a host LU. (For more information about LUs, see “SNA Concepts and Terminology” on page 12.) The LUs coordinate and enforce these data flow rules by monitoring the Send or Receive state at each end of the conversation.

Figure 5-1 on page 81 shows the effect of these data flow rules on the basic format of a Horizon conversation. The numbers in parentheses indicate the sequence of the data flow between the client and server programs.

Most of the Figures in this chapter are arranged to show the logical correlation of the conversation statements issued by each partner program. Arrows show the type of information passed between the partners and often unrealistically suggest that a physical transmission of data accompanies each statement. For more information about how the physical transmission of data is determined, see the discussion of buffering on page 88.



Statement Keyword *State at Completion* and/or *RESULT %variable* and/or *$STATUS Contents*	What Sent	Statement Keyword *State at Completion* and/or *RESULT %variable* and/or *$STATUS Contents*
Client Program		***Server Program***
(1) OPEN PROCESS OUTB ACCEPT *state=Send* . .		(5) OPEN PROCESS INB *state=Receive* . .
(2) SEND *state=Send* . .	data →	(6) RECEIVE *RESULT=DATA* *state=Receive* .
(3) SEND *state=Send* . .	data →	(7) RECEIVE *RESULT=DATA* *state=Receive* .
(4) RECEIVE (waiting)	change-direction →	(8) RECEIVE *RESULT=SEND* *state=Send* . . .
(11) *RESULT=DATA* *state=Receive* . .	← data	(9) SEND *state=Send* . .
(12) RECEIVE *$STATUS=4* *state=Close* .	← close indicator	(10) CLOSE PROCESS *state=Reset*
(13) CLOSE PROCESS *state=Reset*		

Figure 5-1 Conversation Data Flow Illustration

Following is a brief description of the steps in parentheses in Figure 5-1:

Step(s)	Task
(1)	The client program allocates a conversation with a server program. After the OPEN PROCESS completes, the client is in Send state and the server is in Receive state.

Step(s)	Task
(2)-(3)	The client program sends two messages to the server program. The client program then issues a RECEIVE statement.
(4)	The RECEIVE statement puts the client program in Receive state.
(5)-(8)	The server program receives (issues RECEIVE statements) and the host LU returns data to it. When the LU encounters the “change-direction” indicator, it places the server program in Send state, and it notifies the server program by setting its RESULT %variable to “SEND.”
(9)-(10)	The server program sends one message and closes (with CLOSE PROCESS) its side of the conversation.
(11)-(12)	The client program receives until it has received all the messages the server sent. When the LU encounters the “close” indicator, it notifies the client program by setting $STATUS to 4.
(13)	The client program closes its side of the conversation.

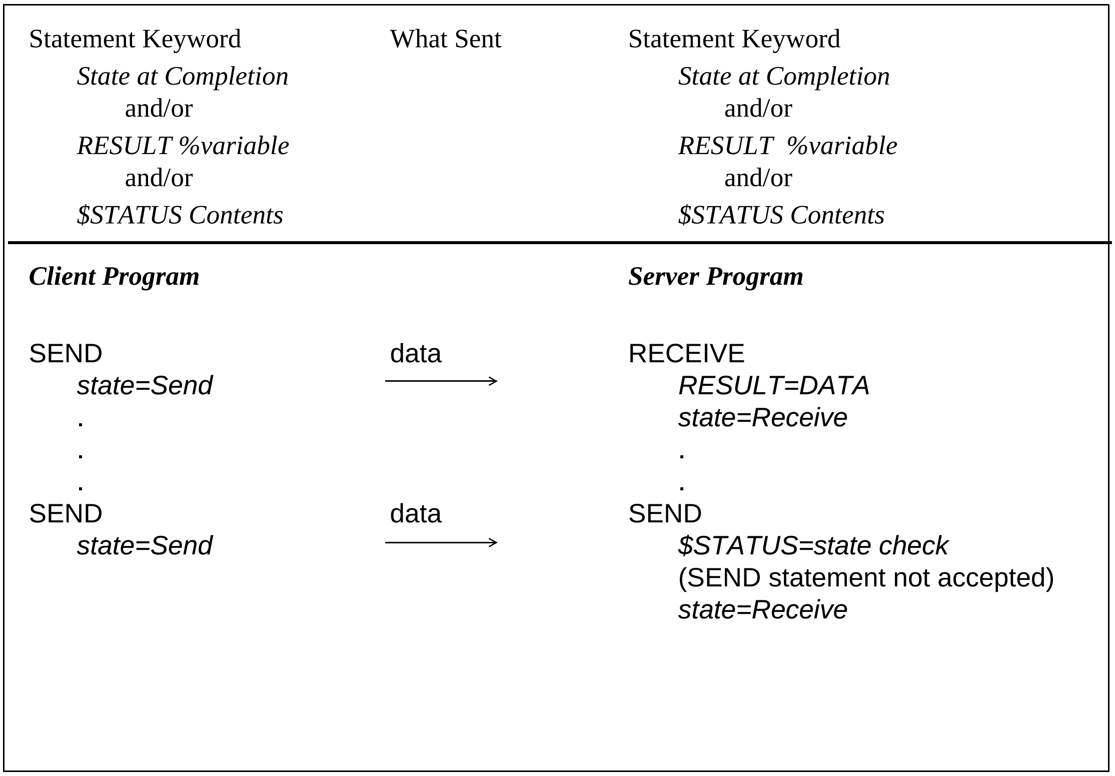
Enforcing direction of data flow

A program puts itself into another state by issuing the appropriate statement during a conversation. For example, a program in Send state changes to Receive state by issuing a RECEIVE statement. The program thereby surrenders the right to send information; it must receive until its partner gives up the right to send.

As shown in Figure 5-2 on page 83, however, a program in Receive state cannot regain send rights by issuing a SEND statement. Issuing SEND is not allowed in Receive state (see Appendix A). If the program issues a SEND statement while in Receive state, it receives a ***state check*** from its LU in the form of non-zero $STATUS and $STATUSD codes. (These codes are set by the LU as completion codes following each SOUL statement issued.)

A state check means that a program is out of synchronization with its partner because of an application program error in one or both partners. A program is issuing a statement while it is in a state in which that statement is not allowed. By issuing state checks, the LUs enforce the conversation data flow rules.

After a state check indication, you can issue a QUERY PROCESS statement to determine the correct conversation state. Then you can terminate the conversation and fix one or both of the partner programs.



Statement Keyword	What Sent	Statement Keyword
State at Completion		*State at Completion*
and/or		and/or
RESULT %variable		*RESULT %variable*
and/or		and/or
$STATUS Contents		*$STATUS Contents*
Client Program		***Server Program***
SEND	data	RECEIVE
state=Send	→	*RESULT=DATA*
.		*state=Receive*
.		.
.		.
SEND	data	SEND
state=Send	→	*$STATUS=state check*
		(SEND statement not accepted)
		state=Receive

Figure 5-2 Conversation State Check

Following the RESULT %variable

The RESULT parameter of the RECEIVE statement indicates to a program in Receive state that its partner has surrendered the right to send. Upon completion of a RECEIVE, the RESULT %variable specified in the RECEIVE contains a character string indicating what was just received (see Steps 6-8 in Figure 5-1 on page 81).

If RESULT contains the string “DATA”, the receiving side knows that it has received information from its partner and should process it accordingly. If RESULT contains the string “SEND”, the receiving side knows that its partner is surrendering the right to send. The receiving program can begin to send data.

Unless it intends to interrupt the sender, the receiving side never acts on its own. It follows the directions in the RESULT %variable until it is told to go into Send state. Once in Send state, it controls the conversation.

Interrupting the Sender

During a conversation, a conversation partner in Receive state can use one of two statements to interrupt the sending side:

- SEND ERROR
- SIGNAL PROCESS

Using SEND ERROR to interrupt the sender

If the receiving side detects an error in the information received, such as the arrival of incorrect data, an “out of space” condition, or an application logic error, it may force a reversal in the direction of data flow by issuing a SEND ERROR statement.

SEND ERROR transmits a negative response to the partner and then discards all not-yet-received information sent by the partner. No further information is presented to the transaction program. Upon receipt of the negative response, the remote LU switches the direction of data flow. The program that issued the SEND ERROR changes into Send state and its partner changes into Receive state (see Figure 5-3 on page 85).

Figure 5-3 shows SEND ERROR used in a conversation.

Statement Keyword	What Sent	Statement Keyword
State at Completion and/or *RESULT %variable* and/or *$STATUS Contents*		*State at Completion* and/or *RESULT %variable* and/or *$STATUS Contents*
Client Program		***Server Program***
. . .		. . .
SEND *$STATUS=0* *state=Send* . . .	data →	RECEIVE *RESULT=DATA* (write to sequential file) . .
SEND *$STATUS=0* *state=Send*	data →	RECEIVE *RESULT=DATA* (write to sequential file; file overflow condition detected) . .
SEND . . *$STATUS=error* *state=Receive* .	data → (discarded) ← error	SEND ERROR . *state=Send* . .
RECEIVE *state=Receive* *RESULT=DATA* .	← data ("file full")	SEND "file full" *state=Send* . .
RECEIVE *$STATUS=4* *state=Close* .	← close indicator	CLOSE PROCESS *state=Reset*
CLOSE PROCESS *state=Reset*		
(Report error to end user: "file full, file transfer failed")		

Figure 5-3 Using SEND ERROR in a File Transfer Application

Using SIGNAL PROCESS to interrupt the sender

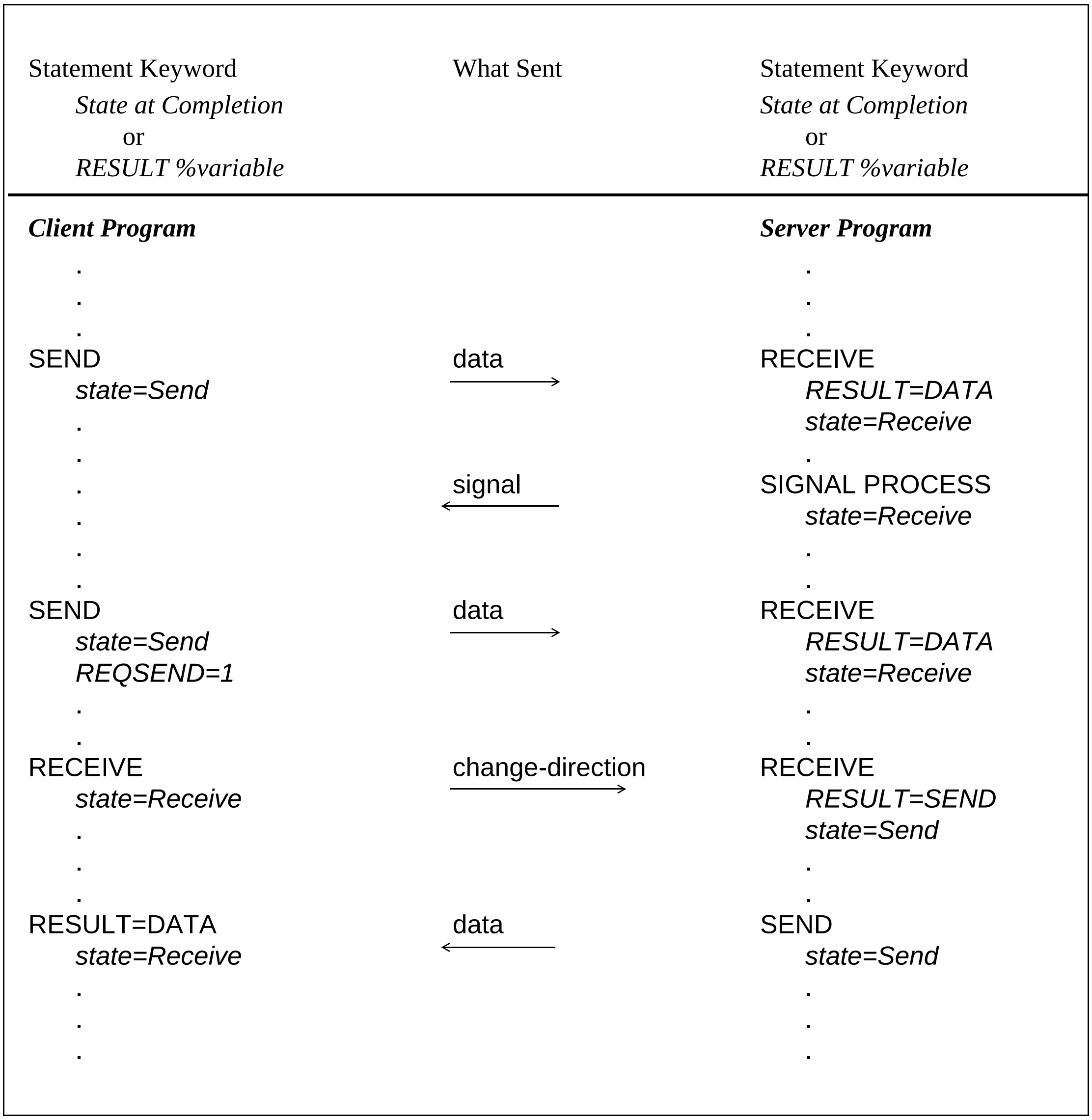
If the receiving side of a conversation wants to interrupt its partner without causing an error condition, it can use the SIGNAL PROCESS statement.

Issuing SIGNAL PROCESS does not put the issuing program into Send state or discard any data that is already in transit from the sending partner. SIGNAL PROCESS merely transmits a desire to send to the partner.

The partner program is notified about this desire to send at the completion of its next SEND statement: its host LU sets a value of 1 in the %variable specified in the REQSEND parameter of SEND. The partner does not have to pay attention to this signal. The issuing program must continue to receive until the partner surrenders the right to send by issuing the RECEIVE statement.

Typically, a sending program knows whether its partner is coded to issue a SIGNAL PROCESS, and only checks the REQSEND %variable if the conversation is designed to use this feature.

Figure 5-4 shows SIGNAL PROCESS used in a conversation.



Statement Keyword *State at Completion* or *RESULT %variable*	What Sent	Statement Keyword *State at Completion* or *RESULT %variable*
Client Program		***Server Program***
. . .		. . .
SEND *state=Send*	data →	RECEIVE *RESULT=DATA* *state=Receive*
. . . . . .	signal ←	. SIGNAL PROCESS *state=Receive* . .
SEND *state=Send* *REQSEND=1* . .	data →	RECEIVE *RESULT=DATA* *state=Receive* . .
RECEIVE *state=Receive* . . .	change-direction →	RECEIVE *RESULT=SEND* *state=Send* . .
RESULT=DATA *state=Receive* . . .	data ←	SEND *state=Send* . . .

Figure 5-4 Using SIGNAL PROCESS

Confirming Receipt of Data

Information is typically passed back and forth during a conversation without an indication of whether it has been received by the intended partner. Sometimes, however, before it can continue with the transaction, one side of a conversation needs confirmation that what it sent was received. The CONFIRM and CONFIRMED statements perform this function.

The sending side issues the CONFIRM statement. Any buffered data is shipped to the partner along with the indication that a confirmation is expected. The receiving side must then issue a CONFIRMED or a SEND ERROR statement before the sending side can continue.

Figure 5-5 is a conversation excerpt showing the use of CONFIRM and CONFIRMED*.*

Statement Keyword *State at Completion* or *RESULT %variable*	What Sent	Statement Keyword *State at Completion* or *RESULT %variable*
Client Program		***Server Program***
SEND *state=Send* . . .	data →	RECEIVE *RESULT=DATA* *state=Receive* . .
CONFIRM (waiting) (return here)	confirmation request →	RECEIVE *RESULT=CONFIRM* *state=Confirm* . . Do what is necessary before confirming (like closing a sequential file to assure that all data is written to non-volatile storage). . .
$STATUS=0 *state=Send*	confirmation indicator ←	CONFIRMED *state=Receive*
		or
$STATUS=error *state=Receive*	error indicator ←	SEND ERROR *state=Send*

Figure 5-5 Confirming Receipt of Data

Buffering and Shipping Conversation Information

This section describes how the LUs of conversation partner programs buffer and ship conversation information from partner to partner.

The Horizon buffering process is such that issuing a SEND statement doesn’t necessarily cause data to be sent immediately. This introduces a potentially confusing interval between the first SEND statement issued and the eventual confirmation (or not) of the receipt of the data sent. Also, since a SEND may not cause data to be shipped immediately, the application programmer must know the statements that do cause immediate shipment.

How the LUs buffer and ship data

Figure 5-6 on page 90 shows the larger context for the conversation depicted in Figure 5-1. The conversation between the partner programs, Program A and Program B, uses a session established between their respective LUs, M204A and M204B.

The LUs buffer data and handle statement processing for the application programs. While a statement issued by an application program is being executed by an LU, the application program’s processing is suspended. Application program processing continues when the LU returns control to the application program.

When a program issues a SEND statement, the data sent is stored in the conversation buffer in that program’s LU. The contents of the buffer are held until it is full or until the application program issues a statement that explicitly causes the LU to send, or ***flush***, the buffer. The statements that can cause an immediate buffer flush are discussed later in this section.

The contents of the sending LU’s conversation buffer is shipped over the session to the partner LU’s conversation buffer, where it is stored until the partner program issues a statement that receives data.

Large blocks of data are broken into multiple network conversation buffers on the sending side and reassembled on the receiving side.

Figure 5-6 shows how the LUs buffer and ship data. This depiction of the physical transmission of conversation information complements the preceding Figures in this chapter. The earlier Figures emphasize the logical correlation of conversation statements.

In Figure 5-6, the data from Program A’s three SEND statements is stored in the conversation buffer in LU M204A. Program A’s RECEIVE statement causes LU M204A to ship the conversation buffer over the session to LU M204B’s conversation buffer. The buffer’s data is staged in a “receive buffer” in LU M204B before being passed through to the conversation buffer.

The data is stored in LU M204B’s conversation buffer until Program B receives it with RECEIVE statements. LU M204A appends a change-direction indicator to the buffered information when Program A issues a RECEIVE statement.

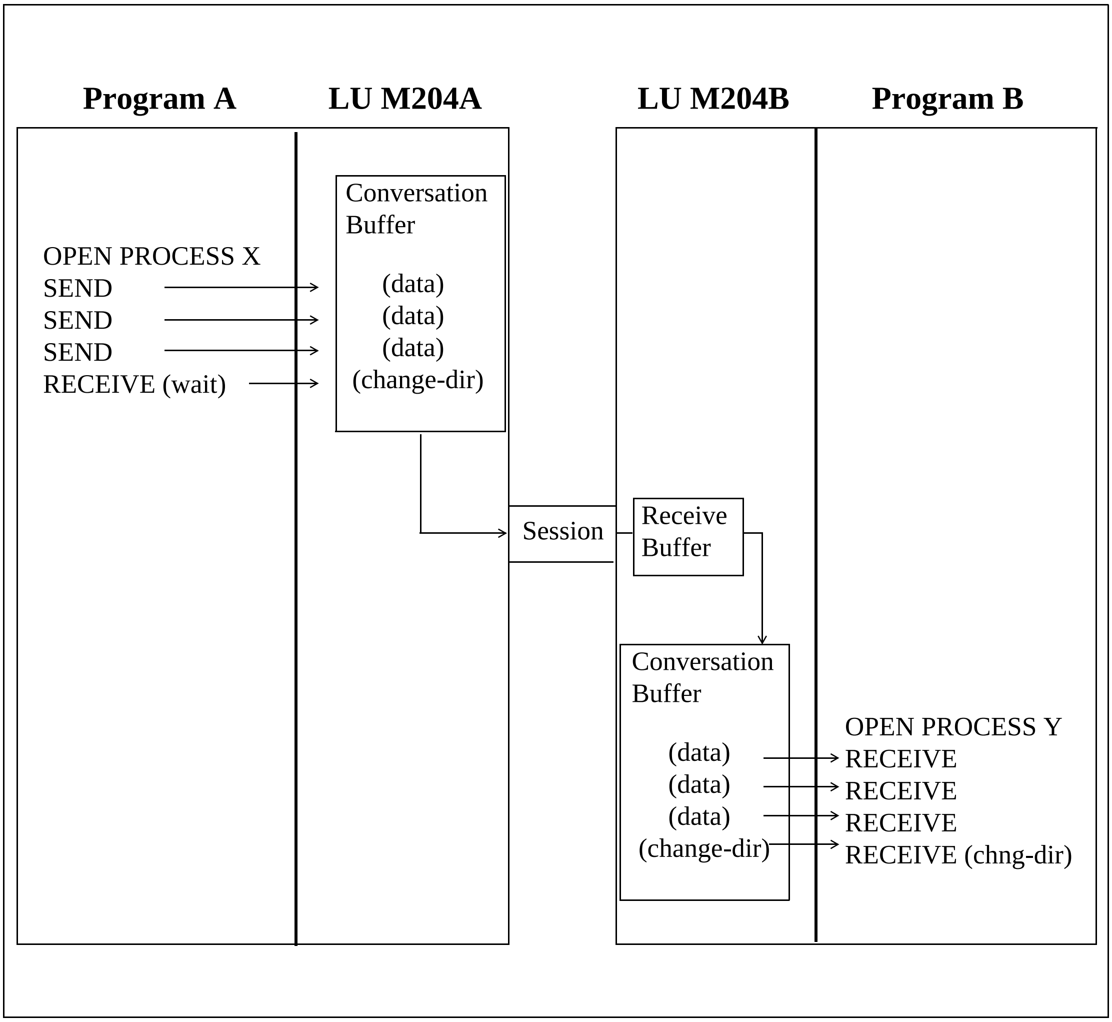


Figure 5-6 Buffering Data

Four statements cause immediate buffer flushing

The following statements can cause an immediate buffer flush:

- RECEIVE

  When issued in Send state, RECEIVE indicates to the host LU that the application program intends to change the conversation direction. The LU then sends all buffered data to the partner.

- CONFIRM

CONFIRM indicates that the application program expects a confirmation from its partner after all data that has been sent up to that point is physically shipped.

- FLUSH PROCESS

  FLUSH PROCESS is used by an application program that needs to force a physical send. FLUSH PROCESS could be used, for example, if a program that sends data only intermittently to its partner needs to have that data physically sent immediately, rather than be buffered to conserve network utilization.

- INVITE

  Issued in Send state, INVITE, like RECEIVE, indicates to the host LU that the application program intends to change the conversation direction. The LU then sends all buffered data to the partner. Unlike RECEIVE, the issuing program does not wait for a reply after the buffer is sent.

Buffering delays error and data-receipt notification

Because the LU buffers data, not always shipping it immediately, certain errors (especially OPEN PROCESS errors) do not reflect back to the client program until a buffer is actually shipped through the network. Depending on how the conversation is constructed, the client program can issue an OPEN PROCESS statement and many SEND statements before it receives an OPEN PROCESS error condition.

Note: A good completion code following a SEND statement is only an indication that the data to be transmitted has been accepted by Model 204. It does not tell the program whether that data has been received by the conversation partner. Before taking any action that depends on the successful arrival of sent data, you should issue a CONFIRM, which forces the data to be physically sent and which requires an acknowledgment from the other side.

Communicating with Multiple Partners Concurrently

Typically a partner program waits until information is received from the remote partner before proceeding to the next statement in the program. This serial processing is satisfactory for applications that depend on input from a specific partner before they can continue, but restricts the ability to communicate efficiently with multiple concurrent partners: regardless of the number of concurrent conversations the program has, the program must request and wait for data from one partner at a time before soliciting data from another partner.

Horizon also supports applications in which a local program sends a request for data to multiple partners and processes their replies in any order. The local program invites its partners to send it data, then either waits for the partners to respond or continues executing other tasks until its partners respond.

Inviting replies

The INVITE statement causes residual data in the send buffer to be flushed to the partner along with a Change Direction indicator. This places the sender in Receive state and the partner in Send state, like issuing the RECEIVE statement. Unlike RECEIVE, INVITE does not cause the program to wait for a reply. Instead, it continues to execute. Subsequent INVITE statements can therefore be issued to other partners before processing of replies begins.

Detecting replies

To detect whether and from whom a reply has arrived, the application requires a WAIT FOR RECEIPT or a TEST RECEIPT statement following INVITE. Both WAIT FOR RECEIPT and TEST RECEIPT permit the application to detect a reply from a specific conversation ID or from any of the partner conversations or processes with outstanding invitations.

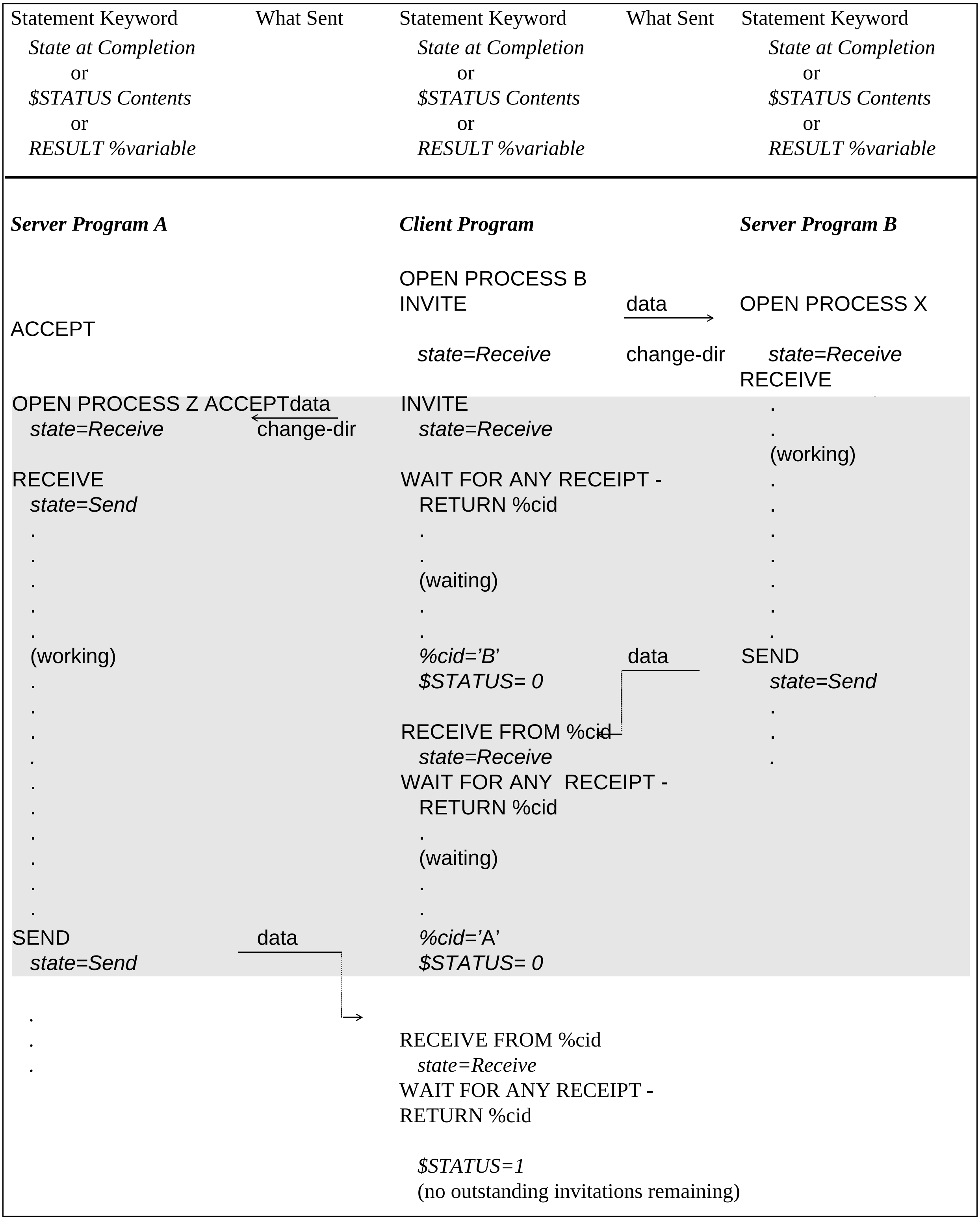
WAIT FOR RECEIPT suspends further running of the program until the specified reply is received. TEST RECEIPT detects the specified reply but does not stop the program from running. Situations for which WAIT FOR RECEIPT and TEST RECEIPT are suitable are described below.

Awaiting requests for data from multiple partners

If your application requires data from multiple partners to complete a task before the program can do further processing, use the WAIT FOR RECEIPT statement after INVITE. The WAIT FOR RECEIPT statement suspends execution of your program until a reply is received from the specified remote partner(s). Your program awakens when a specified reply has arrived. The RECEIVE statement then receives the data into program variables.

Figure 5-5 on page 88 is an example of a SOUL program in which the client program initiates tasks at both server program A and server program B. The client waits while the servers process the request. The $STATUS=0 code following the client’s WAIT FOR ANY RECEIPT statement tells the client to RECEIVE the data from server program B. The $STATUS=0 code following the client’s next WAIT FOR ANY RECEIPT statement tells the client to RECEIVE the data from server program A.

The shaded area in Figure 5-7 indicates where the client program and the server programs are executing concurrently.



Statement Keyword *State at Completion* or *$STATUS Contents* or *RESULT %variable*	What Sent	Statement Keyword *State at Completion* or *$STATUS Contents* or *RESULT %variable*	What Sent	Statement Keyword *State at Completion* or *$STATUS Contents* or *RESULT %variable*
Server Program A		***Client Program***		***Server Program B***
		OPEN PROCESS B		
		INVITE	data	OPEN PROCESS X
ACCEPT				
		state=Receive	change-dir	*state=Receive*
				RECEIVE
OPEN PROCESS Z ACCEPT	data	INVITE		.
state=Receive	change-dir	*state=Receive*		.
				(working)
RECEIVE		WAIT FOR ANY RECEIPT -		.
state=Send		RETURN %cid		.
.		.		.
.		.		.
.		(waiting)		.
.		.		.
.		.		.
(working)		*%cid='B'*	data	SEND
.		*$STATUS= 0*		*state=Send*
.				.
.		RECEIVE FROM %cid		.
.		*state=Receive*		.
.		WAIT FOR ANY RECEIPT -		
.		RETURN %cid		
.		.		
.		(waiting)		
.		.		
.		.		
SEND	data	*%cid=*'A'		
state=Send		*$STATUS= 0*		
.				
.		RECEIVE FROM %cid		
.		*state=Receive*		
		WAIT FOR ANY RECEIPT - RETURN %cid		
		$STATUS=1 (no outstanding invitations remaining)		

Figure 5-7 Waiting for receipt of data

Initiating a background task

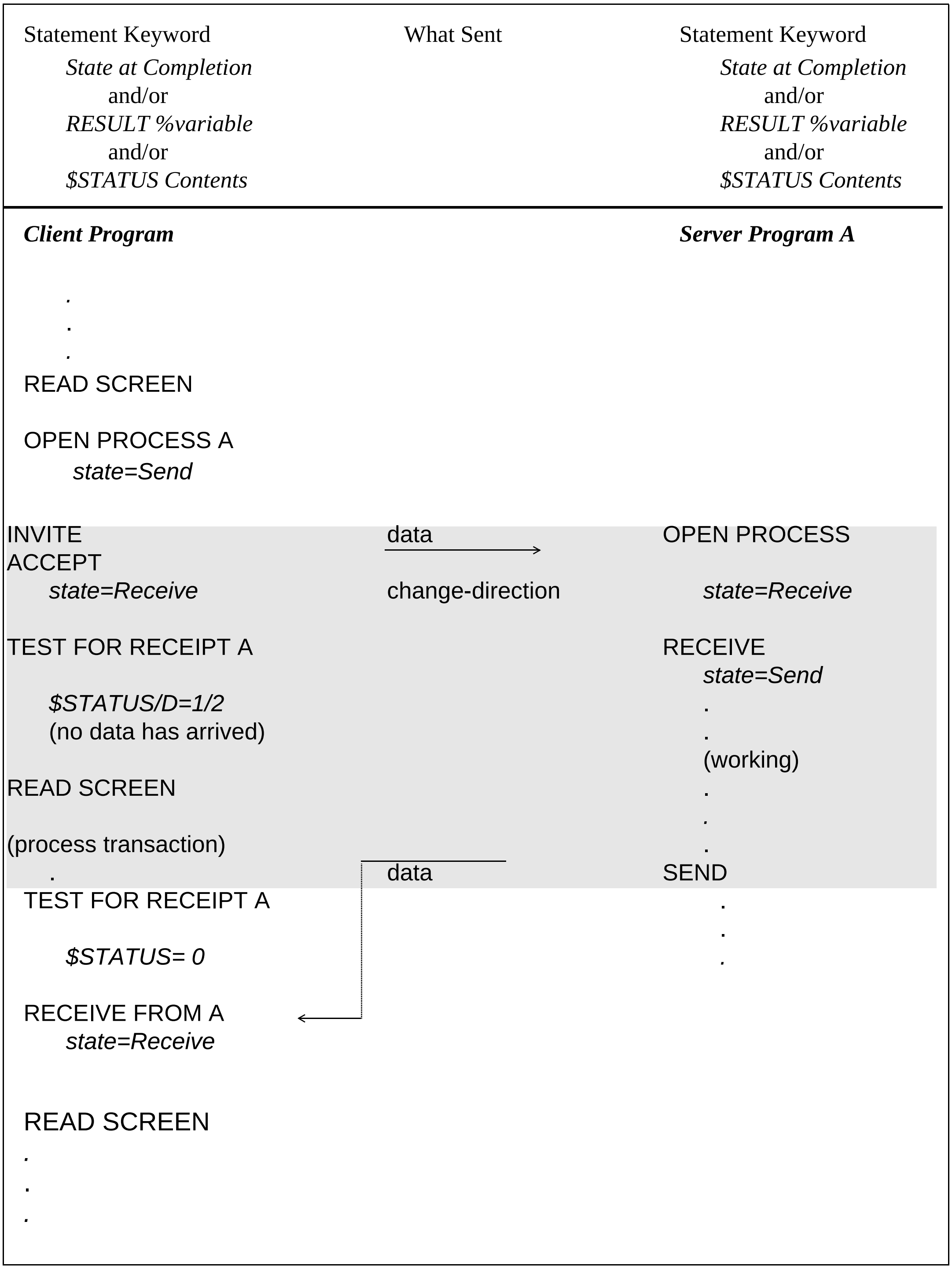
You can use the INVITE statement followed by TEST RECEIPT to have a partner process serve as a background task. Since the TEST RECEIPT statement does not suspend execution of your program, you can invite a partner program to perform a task and can continue to perform other tasks until a reply is received from the partner. When TEST RECEIPT $STATUS codes indicate that a reply has arrived, the RECEIVE statement puts the data into program variables.

The application subsystem performing the background task can execute both command level commands and SOUL procedures.

Figure 5-8 on page 95 is an example of a SOUL program in which the client program initiates a task at server program A and continues to run while server program A is processing the task request. The client periodically (in this case, each time it is about to use READ SCREEN) issues a TEST FOR RECEIPT A statement. When the $STATUS code indicates that no reply has been received ($STATUS/D=1/2), the client program continues with other work. When the $STATUS code indicates that server program A has replied ($STATUS=0), the client program RECEIVEs the data.

The shaded area in the figure indicates where the client program and the server programs are executing concurrently.

With the method shown in Figure 5-8, you can initiate multiple background tasks and can test for replies from invited partners.



Statement Keyword *State at Completion* and/or *RESULT %variable* and/or *$STATUS Contents*	What Sent	Statement Keyword *State at Completion* and/or *RESULT %variable* and/or *$STATUS Contents*
Client Program		***Server Program A***
. . .		
READ SCREEN		
OPEN PROCESS A *state=Send*		
INVITE	data →	OPEN PROCESS
ACCEPT		
state=Receive	change-direction	*state=Receive*
TEST FOR RECEIPT A		RECEIVE *state=Send*
$STATUS/D=1/2 (no data has arrived)		. . (working)
READ SCREEN		. .
(process transaction)		.
.	data	SEND
TEST FOR RECEIPT A		. . .
$STATUS= 0		
RECEIVE FROM A *state=Receive*	←	
READ SCREEN . . .		

Figure 5-8   Testing for receipt of data

6

Horizon SOUL Interface

Overview

This section describes individually the SOUL statements used by Horizon. The description of each statement contains usage information, complete syntax, and conversation state dynamics.

Dependence on conversation state

Remember that the Horizon SOUL statements that can be issued by a process are dependent on the state of the particular conversation. This state can be determined by using the QUERY PROCESS statement, which is described in this section.

Error checking

Error conditions encountered during the execution of the Horizon SOUL statements are not displayed on the terminal. The SOUL request should check for errors by using the $STATUS and $STATUSD functions. The text of the most recently issued error message can be retrieved by using the $ERRMSG function.

CLOSE PROCESS Statement

Function The CLOSE PROCESS statement deallocates the specified conversation from the process.

Syntax The CLOSE PROCESS format follows:

```
CLOSE PROCESS {cid | processname | %variable}
      [SYNCLEVEL | FLUSH | CONFIRM | ERROR | %variable]
```

Where:

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation.
- If no CID value is specified on the OPEN PROCESS statement, *processname* should be used. If a CID value is specified on the OPEN PROCESS, you also must specify that value here on the CLOSE PROCESS statement.
- *SYNCLEVEL* specifies that the actions taken depend on whether CONFIRM or NOCONFIRM is specified on the DEFINE PROCESS command:
  - If CONFIRM is specified, the actions indicated by CLOSE PROCESS CONFIRM are taken.
  - If NOCONFIRM is specified, the actions indicated by CLOSE PROCESS FLUSH are taken.

  SYNCLEVEL is the default.
- *FLUSH* specifies that any data buffered by preceding SEND statements is transmitted to the remote partner before the conversation is deallocated.
- *CONFIRM* specifies that a CONFIRM request is sent to the remote partner before the conversation is deallocated. CONFIRM also implies a buffer flush.
- *ERROR* specifies that the conversation is terminated abnormally.
- *%variable* contains “SYNCLEVEL,” “FLUSH,” “CONFIRM,” or “ERROR.”

Conversation state

Valid conversation states

State	Statement
Send state	For CLOSE PROCESS with SYNCLEVEL, CONFIRM, or FLUSH
Send, Receive, or Confirm state	For CLOSE PROCESS with ERROR

Status codes

S /	SD	New State	Description
0	0	Reset	Normal completion.
2	2	Receive	SEND ERROR statement issued by partner.
4	0	Reset	Partner process closed conversation normally.
4	1	Reset	Partner process ended abnormally.
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meanings of the individual $STATUSD codes.
10-99		Close	Resource allocation failures (see Table 8-1).

CONFIRM Statement

Function

CONFIRM sends a confirmation request to the remote partner. After issuing CONFIRM, the sending program waits for the receiver to issue CONFIRMED or SEND ERROR. This exchange allows the local and remote partners to synchronize their processing. CONFIRM causes the conversation buffer, which contains all accumulated messages, to be flushed.

Syntax

The CONFIRM statement format follows:

```
CONFIRM {cid | processname | %variable} REQSEND %variable
```

Where:

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation.
- If no CID value is specified on the OPEN PROCESS statement, *processname* should be used. If a CID value is specified on the OPEN PROCESS, you also must specify that value here on the CONFIRM statement.

- *REQSEND %variable* is used to detect a signal from the partner. While the conversation is in Send state, a SIGNAL PROCESS (REQUEST_TO_SEND) statement from the conversation partner can be received, indicating that the partner wishes to enter Send state.

  Reception of a SIGNAL from the partner is indicated in the REQSEND %variable:

  – A value of 1 indicates a SIGNAL was received.

  – A value of 0 indicates no SIGNAL was received.

Conversation state Valid conversation state: `Send`

Status codes

S /	SD	New State	Description
0	0	Send	Normal completion.
2	2	Receive	SEND ERROR statement issued by partner.
3	3	no change	State check.
4	1	Close	Partner process ended abnormally.
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meanings of the individual $STATUSD codes.
10-99		Close	Resource allocation failures (see Table 8-1).

CONFIRMED Statement

Function The CONFIRMED statement sends a positive confirmation to the remote partner that the last statement was executed successfully. This is only issued in response to a CONFIRM request.

Syntax The syntax of the CONFIRMED statement follows:

```
CONFIRMED {cid | processname | %variable}
```

Where:

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation.

- If no CID value is specified on the OPEN PROCESS statement, *processname* should be used. If a CID value is specified on the OPEN PROCESS, you also must specify that value here on the CONFIRMED statement.

Conversation state

Valid conversation state: `Confirm`

Status codes

S /	SD	New State	Description
0	0	Receive	Normal completion when issued in Confirm state.
		Send	Normal completion when issued in Confirm Send state.
		Close	Normal completion when issued in Confirm Close state.
3	3	no change	State check.
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meaning of each $STATUSD code.
10-99		Close	Resource allocation failures (see Table 8-1).

FLUSH PROCESS Statement

Function

The FLUSH PROCESS statement causes the conversation buffer to be sent to the remote partner.

Syntax

The FLUSH PROCESS format follows:

```
FLUSH PROCESS {cid | processname | %variable}
```

Where:

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation.
- If no CID value is specified on the OPEN PROCESS statement, *processname* should be used.   If a CID value is specified on the OPEN PROCESS, you also must specify that value here on the FLUSH PROCESS statement.

Conversation state

Valid conversation state: `Send`

Status codes

S /	SD	New State	Description
0	0	Send	Normal completion.
2	2	Receive	SEND ERROR statement issued by partner.
3	3	no change	State check.
4	1	Close	Partner process ended abnormally.
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meaning of each $STATUSD code.
10-99		Close	Resource allocation failures (see Table 8-1).

INVITE Statement

Function

The INVITE statement permits a program to request data from one or more partners simultaneously and allows the issuing program to continue to run while its requests for data are being processed. The INVITE statement causes the send buffer to be flushed to the partner, and it places the partner in Send state, enabling the partner to send data when it is ready. After INVITE is issued, the issuing program continues to execute.

The program that issues INVITE has the option of requesting that its partner send confirmation before entering Send state.

Usage

The INVITE statement is used to change the direction of the conversation from Send to Receive state. The RECEIVE statement is used to accept the data transmitted by the partner.

Following each INVITE statement you may issue multiple RECEIVE statements to accept data sent in reply to the INVITE. However, *issuing consecutive INVITEs to the same partner is an error*. Since INVITE may be issued only in Send state, and since the first INVITE statement changes the conversation state from Send to Receive, a second INVITE results in a state check. When the “inviting” partner next receives a send indication from its partner, it again enters Send state and may again issue INVITE.

Syntax

The INVITE statement format follows:

```
INVITE {cid|processname|%variable}
 [SYNCLEVEL|FLUSH|CONFIRM]
```

Where:

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation. If no CID value is specified on the OPEN PROCESS statement, use the process name.

- *SYNCLEVEL*, the default, specifies that the actions taken depend on whether CONFIRM or NOCONFIRM is specified on the DEFINE PROCESS command for the program that issues this INVITE:
  - If CONFIRM is specified, the actions described below for INVITE CONFIRM are taken.
  - If NOCONFIRM is specified, the actions described below for INVITE FLUSH are taken.
- *FLUSH* causes the LU to flush the contents of the send buffer, sending with it a change-direction indicator. The remote partner enters Send state.
- *CONFIRM* causes the local LU to flush the contents of the send buffer along with a confirmation request and a change-direction indicator. The remote partner enters Confirm state and must issue an appropriate verb (CONFIRMED, SEND ERROR, or CLOSE). If it issues CONFIRMED or SEND ERROR, it then enters Send state.

  INVITE CONFIRM is valid only when the conversation's synchronization level is CONFIRM.

Conversation state

Valid conversation state: Send

Status codes and state transitions

S /	SD	New State	Description
0	0	Receive	OK
2	2	Receive	Partner responded to INVITE CONFIRM with SEND ERROR
3	3	no change	State check (issued in wrong conversation state)
4	1	Close	Partner process ended abnormally
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meaning of each $STATUSD code.
10-99		Close	Resource allocation failures (see Table 8-1).

OPEN PROCESS Statement

Function

The OPEN PROCESS statement allocates local resources necessary for a conversation. These resources include local storage as well as a session connecting the local Model 204 system to the remote partner. If a non-busy session is available, it is used; otherwise an attempt to activate a new session is made if session limits have not been reached.

There are two distinct uses of the OPEN PROCESS statement:

- For a client process, it allocates an outbound conversation.

  An outbound conversation is identified by the absence of the ACCEPT clause. Allocation involves creating control blocks, getting control of a session for the conversation, and requesting the initiation of the remote process with optional initialization parameters.

- For a server process, it identifies an inbound conversation.

  An inbound OPEN PROCESS statement, identified by OPEN PROCESS with the ACCEPT clause, may be issued only by a server process. It receives any initialization parameters passed to it during the initialization of the conversation by the client process.

In addition, the OPEN PROCESS statement establishes the conversation ID (CID) for both types of conversations.

Usage

A conversation begins when an OPEN PROCESS statement is issued in a SOUL request. A conversation ends when one of the following happens:

- The request issues the CLOSE PROCESS statement.
- The end user’s thread returns to command level.
- The request issues the END statement (TPROCESS only).
- The request is cancelled.
- The thread is restarted.
- The network connection is severed by the Model 204 system administrator who issues the CLOSE LINK command.
- The communication path (session) is broken by some external event beyond the control of Model 204.

Syntax

The OPEN PROCESS format follows:

```
OPEN PROCESS {processname] %variable}
[CID {name | %variable}]

Client process (outbound) parameters:

[AT {destination | %variable}]
[WITH]
[USERID {%variable | ’string’}]
[PASSWORD {%variable | ’string’}]
[{ACCOUNT | PROFILE} {%variable | ’string’}]
[INITIAL {DATA ’string’ | DATA %variable
          | IMAGE image}...]
```

```
Server process (inbound) parameters:

ACCEPT
[INITIAL {DATA %variable | IMAGE image}...]
```

Where:

- *processname* depends on the program that issues the OPEN PROCESS:
  - For a client (outbound) program: *processname* is the name of a client process specified in a client DEFINE PROCESS command.
  - For a server (inbound) program: *processname* is any name chosen by the SOUL programmer.

Note: Names beginning with CCA are specifically disallowed from use in the OPEN PROCESS statement.

- *CID* is the ID of a conversation for subsequent Horizon SOUL statements. If CID is not specified, processname is used as the ID. However, CID must be specified if the user has two or more active conversations with the same processname. If CID is specified, subsequent conversation statements must also specify the same CID value.
- *AT* specifies the symbolic destination for a remote partner. The *destination* name must correspond to a symbolic name specified on the DESTINATION parameter of the DEFINE PROCESS command for *processname*. Refer to the DEFINE PROCESS command client (outbound) options. If not specified, the first processgroup of the DESTINATION parameter is used.
- AT may **not** be specified if there is only one processgroup in the DESTINATION parameter.
- *USERID* is the user ID to be logged in at the server node on behalf of this OPEN PROCESS. If this parameter is omitted, Model 204 transmits the current user ID. If the local user on the client system is not logged in, Model 204 transmits no user ID. If this parameter is coded, the DEFINE PROCESS command for processname must specify UIDSOURCE=OPEN.
- *PASSWORD* is the password to be used for login at the server node. If USERID is coded, this parameter is required. If USERID is omitted, this parameter is used if coded and if a current user ID exists for the thread.
- *ACCOUNT* and *PROFILE* are synonyms and are mutually exclusive. They may only be coded if the DEFINE PROCESS command for processname specifies the OPEN option. They behave the same way the USERID parameter does.
- *INITIAL* depends on the program that issues the OPEN PROCESS:
  - For a client (outbound) program: INITIAL specifies the Program Initialization Parameters (PIP data) to be passed to the remote partner
  - For a server (inbound) program: INITIAL specifies the SOUL areas used to accept the Program Initialization Parameters

The total length of the PIP data must be less than the DATALEN value specified on the DEFINE PROCESS command for processname.

Conversation state Valid conversation state: `Reset`

Status codes

S /	SD	New State	Description
0	0	Send	Normal completion when issued by client (outbound) program.
		Receive	Normal completion when issued by server (inbound) program.
2	2	Receive	SEND ERROR statement issued by partner.
4	1	Close	Partner process ended abnormally.
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meanings of the individual $STATUSD codes.
10-99		Close	Resource allocation failures (See Table 8-1).

QUERY PROCESS Statement

Function QUERY PROCESS allows the SOUL programmer to retrieve information from the host LU about one or more characteristics of the conversation.

Syntax Following the choice of *cid, processname,* or *%variable* is a list of parameter/%variable pairs. The parameters correspond to the conversation characteristics about which information can be returned by the host LU.

After a program issues a QUERY PROCESS statement with one or more parameters and accompanying %variables, the LU returns the value of the indicated conversation characteristic, storing it in the %variable following the parameter. A compilation error occurs if the type or length of any %variable is invalid.

The QUERY PROCESS format follows:

```
QUERY PROCESS {cid | processname | %variable}
```

Parameters (include at least one):

```
STATE %variable
PROCESSGROUP %variable
REMOTEID %variable
```

```
SYNCLEVEL %variable
MODENAME %variable
```

Where:

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation, if it is open.
- If no CID value is specified on the OPEN PROCESS statement, *processname* should be used. If a CID value is specified on the OPEN PROCESS, you also must specify that value here on the QUERY PROCESS statement.
- *STATE* is the current conversation state of the conversation. The value of STATE is a string of maximum length 7. Value options follow:
  – RESET
  – SEND
  – RECV (to indicate Receive state)
  – CONFIRM
  – CONFSND (to indicate Confirm Send state)
  – CONFCLS (to indicate Confirm Close state)
  – CLOSE

  For more information about conversation states, see “Conversation states” on page 78. For more information about the Confirm Send and Confirm Close states, see the discussion of the RESULT parameter of the RECEIVE statement in “RECEIVE Statement” on page 108.
- *PROCESSGROUP* is the name of the processgroup associated with the conversation. The value is a string of length 8.
- *REMOTEID* is the name of the LU at which the remote partner is located. The value is a string of length 8.
- *SYNCLEVEL* is the level of synchronization processing used for the conversation, that is, whether confirmation processing is allowed. The value is a string of maximum length 9. Values are:
  – CONFIRM
  – NOCONFIRM

  CONFIRM or NOCONFIRM can be set on the DEFINE PROCESS command, described in “Defining a Horizon network” on page 26.
- *MODENAME* is the mode table entry for the session which the conversation is using. The value is a string of length 8.

Conversation state

Valid conversation states:

- Any state: for STATE option

- Any state except Reset state: for other options

Status codes and state transitions

S /	SD	New State	Description
0	0	no change	Normal completion.
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meanings of the individual $STATUSD codes.
10-99		Close	Resource allocation failures (see Table 8-1).

RECEIVE Statement

Function

The RECEIVE statement receives any data or indicator sent by the remote partner.

Syntax

The RECEIVE statement format follows:

```
RECEIVE {IMAGE imagename | %variable} FROM
   {cid | processname | %variable} RESULT %variable
```

Where:

- *imagename* specifies that the receiving area is an image previously defined by an image definition.

  The receiving area can also be a %variable. The data received from the remote partner is assumed to be in string form and is converted to the type of %variable.

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation.

- If no CID value is specified on the OPEN PROCESS statement, *processname* should be used.   If a CID value is specified on the OPEN PROCESS, you also must specify that value here on the RECEIVE statement.

- *RESULT %variable* is used to determine what the process has received. RESULT should only be examined if $STATUS is 0 or 1, since otherwise, nothing is placed in this %variable.

  The RESULT %variable must be a string variable at least 15 characters in length. It can contain only one of these values following a RECEIVE statement:

  – DATA: All of the data issued by one send operation by the remote partner has been received in the %variable or IMAGE. (It may be necessary to IDENTIFY another IMAGE to reference all of the data.)

- DATA TRUNCATED: Part of the data issued by one send operation by the remote partner has been received in the %variable or IMAGE. (It may be necessary to IDENTIFY another IMAGE to reference even the data received.) The data is truncated on the right. To correct this problem, increase the setting of DATALEN on the DEFINE PROCESS-GROUP command.
- SEND: No data was received; the SOUL program may issue a SEND statement.
- CONFIRM: No data was received; the SOUL program may issue a CONFIRMED or SEND ERROR statement; a CONFIRMED statement can return the state to Receive.
- CONFIRM SEND: No data was received. The SOUL program may now issue a CONFIRMED or SEND ERROR statement; a CONFIRMED statement can change the state to Send.
- CONFIRM CLOSE: No data was received. The SOUL program may issue a CONFIRMED or SEND ERROR statement; a CONFIRMED statement can change the state to Reset.

A compilation error results if the type or length of the RESULT %variable is incorrect.

Note: The end of the conversation is indicated by $STATUS=4 and not by a RESULT value.

Conversation state

Valid conversation states: `Send or Receive state`

Status codes and state transitions

<table>
<tr><th>S</th><th>SD</th><th>New State</th><th>Description</th></tr>
<tr><td>0</td><td>0</td><td>Unchanged</td><td>Normal completion.</td></tr>
<tr><td>1</td><td>0</td><td>Check RESULT</td><td>Special completion: no data, check RESULT:
<table>
<tr><th>RESULT %variable</th><th>Meaning</th></tr>
<tr><td>DATA TRUNCATED</td><td>Data received was longer than target variable or DATALEN; remain in Receive state.</td></tr>
<tr><td>DATA INCOMPLETE</td><td>Not used (always truncated)</td></tr>
<tr><td>SEND</td><td>Receiver is now in Send state.</td></tr>
<tr><td>CONFIRM</td><td>Respond with CONFIRMED or SEND ERROR; remain in Receive state.</td></tr>
<tr><td>CONFIRM SEND</td><td>Respond with CONFIRMED or SEND ERROR; enter Send state.</td></tr>
<tr><td>CONFIRM CLOSE</td><td>Respond with CONFIRMED or SEND ERROR; enter Close state.a</td></tr>
</table></td></tr>
<tr><td>2</td><td>2</td><td>Unchanged</td><td>SEND ERROR statement issued by partner.</td></tr>
<tr><td>3</td><td>3</td><td>Unchanged</td><td>State check.</td></tr>
<tr><td>4</td><td>0</td><td>Close</td><td>Partner process closed conversation normally.</td></tr>
<tr><td>4</td><td>1</td><td>Close</td><td>Partner process ended abnormally.</td></tr>
<tr><td>5</td><td>any</td><td>no change</td><td>Parameter check (coding error). See Table 8-1, which lists the meanings of the individual $STATUSD codes.</td></tr>
<tr><td>10-99</td><td></td><td>Close</td><td>Resource allocation failures (see Table 8-1).</td></tr>
</table>

SEND Statement

Function

Initiate transmission of data from an application program to its remote partner.

Syntax

```
SEND [IMAGE imagename |’string’| %variable]
 TO {cid | processname | %variable} [FLUSH | CONFIRM]
 REQSEND %variable
```

Where:

- *imagename* specifies that the object sent is an image previously defined by an image definition.

  The object sent can also be a quoted string or a %variable. A %variable is converted to string form before being sent to the remote partner.

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation.
- If no CID value is specified on the OPEN PROCESS statement, *processname* should be used. If a CID value is specified on the OPEN PROCESS statement, you also must specify it on the SEND statement.
- *FLUSH* is identical to issuing a FLUSH statement following the SEND.
- *CONFIRM* is identical to issuing a Confirm statement following the SEND.
- *REQSEND %variable* is used to detect a signal from the partner. While the conversation is in Send state, a signal may be received, indicating that the remote conversation partner wishes to enter Send state.

  The REQSEND %variable indicates a signal from the partner:

  – A value of 1 indicates a SIGNAL was received.

  – A value of 0 indicates no SIGNAL was received.

Conversation state Valid conversation state: `Send`

Status codes and state transitions

S /	SD	New State	Description
0	0	Send	Normal completion.
2	2	Receive	SEND ERROR statement issued by partner.
3	3	no change	State check.
4	1	Close	Partner process ended abnormally.
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meanings of the individual $STATUSD codes.
10-99		Close	Resource allocation failures (see Table 8-1).

SEND ERROR Statement

Function The SEND ERROR statement sends a negative acknowledgment indicating that the local partner detected an application error.

Syntax The SEND ERROR statement format follows:

```
SEND ERROR TO {cid | processname | %variable}
  REQSEND %variable
```

Where:

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation.
- If no CID value is specified on the OPEN PROCESS statement, *processname* should be used. If a CID value is specified on the OPEN PROCESS, you also must specify that value here on the SEND ERROR statement.
- *REQSEND* %variable is used to detect a signal from the partner. While the conversation is in Send state, a SIGNAL (REQUEST_TO_SEND) may be received, indicating that the remote conversation partner wishes to enter Send state.

  Reception of a signal from the partner is indicated in the REQSEND %variable:

  – A value of 1 indicates a SIGNAL was received.

  – A value of 0 indicates no SIGNAL was received.

Conversation state

Valid conversation states: `Send, Receive, or Confirm state`

Status codes and state transitions

S /	SD	New State	Description
0	0	Send	Normal completion.
2	2	Receive	SEND ERROR statement issued by partner.
3	3	no change	State check.
4	1	Close	Partner process ended abnormally.
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meanings of the individual $STATUSD codes.
10-99		Close	Resource allocation failures (see Table 8-1).

SIGNAL PROCESS Statement

Function

The SIGNAL PROCESS statement notifies the remote partner that the local partner is requesting to enter Send state for the conversation.

Syntax

The SIGNAL PROCESS format follows:

```
SIGNAL PROCESS {cid | processname | %variable}
```

Where:

- *cid* or *processname* or *%variable* is the conversation ID on the OPEN PROCESS statement for this conversation.
- If no CID value is specified on the OPEN PROCESS statement, *processname* should be used. If a CID value is specified on the OPEN PROCESS, you also must specify that value here on the SIGNAL PROCESS statement.

Conversation state information

Valid conversation states: `Receive or Confirm state`

Status codes and state transitions

S /	SD	New State	Description
0	0	no change	Normal completion.
3	3	no change	State check.
5	any	no change	Parameter check (coding error). See Table 8-1, which lists the meanings of the individual $STATUSD codes.
10-99		Close	Resource allocation failures (see Table 8-1).

TEST RECEIPT Statement

Function

The TEST RECEIPT statement determines whether or not a partner process has sent anything in reply to an INVITE statement. Using the TEST RECEIPT statement instead of a WAIT FOR RECEIPT statement allows the issuing program to perform other tasks while the partner is preparing its reply to the INVITE.

The TEST RECEIPT statement permits the application to test for the reply from a specific conversation ID or from any of the partner conversations or processes.

After a reply is detected, the RECEIVE statement is used to receive the data or indicator sent.

Syntax

The TEST RECEIPT statement format follows:

```
TEST [FOR] ANY RECEIPT RETURN %variable
```

```
TEST [FOR] RECEIPT {cid | processname | %variable}
```

Where:

- *TEST ANY RECEIPT* means that the receipt of information for *any one of* the conversations with outstanding INVITEs is sufficient to satisfy the TEST.

%variable is a SOUL string variable into which is stored the CID of the conversation that replied. Only this form of the TEST statement requires the return of identifying information from the partner.

- *TEST RECEIPT* means that the receipt of information from the conversation or partner specified *by cid* or *processname* or *%variable* is sufficient to satisfy the TEST.

  cid or *processname* or *%variable* is the conversation or partner identifier on the OPEN PROCESS statement for this conversation. If no CID value is specified on the OPEN PROCESS statement, use the process name.

Conversation state information

Valid conversation states: Not applicable (N/A). Horizon does no state checking for TEST RECEIPT.

Status codes and state transitions

S /	SD	New State	Description
0	0	N/A	OK (data has arrived from remote partner).
1	1	N/A	No outstanding INVITEs remain.
1	2	N/A	No data has arrived from remote partner.
5	5	N/A	Process not opened (only if cid or process name specified).
5	17	N/A	Cid or process name too long.
10-99		N/A	Resource allocation failures (see Table 8-1).

WAIT FOR RECEIPT Statement

Function

The WAIT FOR RECEIPT statement causes the issuing SOUL program to stop running until a reply is received from the specified remote partner(s). After a reply is detected, the RECEIVE statement is used to receive the data or indicator sent.

The WAIT FOR RECEIPT statement permits the application to wait for a specific conversation ID to respond before it awakens. Or, the application can awaken when any of the conversations or processes respond.

Syntax

The WAIT FOR RECEIPT statement format follows:

```
WAIT [duration] [FOR] ANY RECEIPT RETURN %variable
WAIT [duration] [FOR] RECEIPT
 {cid | processname | %variable}
```

Where:

- *WAIT FOR ANY RECEIPT* causes the program to reawaken upon the receipt of information from **any one of** the conversations with outstanding INVITEs.

  %variable is a SOUL string variable into which is stored the CID of the conversation that replied. Only this form of the WAIT FOR RECEIPT statement requires the return of identifying information from the partner.

- *WAIT FOR RECEIPT* causes the program to reawaken upon the receipt of information from the conversation or partner specified by *cid* or *processname* or *%variable*.

  cid or *processname* or *%variable* is the conversation or partner identifier on the OPEN PROCESS statement for this conversation. If no CID value is specified on the OPEN PROCESS statement, use the process name.

- *duration* = integer | %variable SEC[S]

  duration is an optional parameter that limits the length of time (in seconds) that the WAIT statement waits for the specified receipt to occur. If the specified receipt does not occur before or during the execution of the timed WAIT, the WAIT statement completes with $STATUS=1 and $STATUSD=3.

Note: *duration* has nothing to do with the TIMEOUT parameter of the DEFINE PROCESS command for the conversation. If TIMEOUT is exceeded during execution of a conversation statement, the conversation is immediately terminated abnormally. The timed WAIT allows the programmer a variety of possible consequent actions if duration is exceeded, including abnormal termination of the conversation.

Conversation state

Valid conversation states: Not applicable (N/A). Horizon does no state checking for WAIT FOR RECEIPT.

Status codes and state transitions

S /	SD	New State	Description
0	0	N/A	OK - information has arrived.
1	1	N/A	No outstanding INVITEs to wait for.
1	3	N/A	WAIT duration reached before events complete.
5	any	N/A	Parameter check (coding error). Table 8-1 lists the meanings of the individual $STATUSD codes.
10-99		N/A	Resource allocation failures (see Table 8-1).

7

Horizon User Language (SOUL) Sample Programs

Client Program

The following example is a small client program that establishes a conversation with a partner process, sends a message to the partner, receives messages from the partner, and ends the conversation.

The STAT_CHECK subroutine is used after each HORIZON statement to determine the appropriate subsequent action for the program to take. Subroutine STAT_CHECK is found in “Subroutine to Check Horizon Return Codes” on page 125.

The HORIZON LINK, PROCESSGROUP, and PROCESS definition commands are included as comments at the beginning of the procedure, for convenient reference.

Companion program

“Server Sample” on page 121 has an example of a server program that can communicate with the client program shown here.

Program

```
PROCEDURE CLIENT
BEGIN

******************************************************************
***   S A M P L E  H O R I Z O N  C L I E N T  P R O G R A M   ***
******************************************************************
****************************************
```

```
***      NETWORK DEFINITIONS          ***
*****************************************
*
* DEFINE LINK VIRGINIA WITH SCOPE=SYSTEM LOCALID=FREDBURG  -
*  INBUFSIZE=2048 TRANSPORT=VTAM PROTOCOL=LU62 SESSIONS=4
*
* DEFINE PROCESSGROUP BOULDER WITH SCOPE=SYSTEM OUTLIMIT=4  -
*  INLIMIT=0 REMOTEID=COLORADO LINK=VIRGINIA
*
* DEFINE PROCESS WEEKEND WITH SCOPE=SYSTEM PARTNER=SOMEFUN  -
*  DESTINATION=(BOULDER,FAC,MOAB,ARCHES) DATALEN=500
*
******************************************
***        DATA AND SUBROUTINES        ***
******************************************

 %CID IS STRING LEN 8
 %RECVMSG IS STRING LEN 255
 %RESULT IS STRING LEN 15
 %REQSEND IS FLOAT
 %ADVICE IS FLOAT

 %CID = ’MADAME’

 INCLUDE STAT_CHECK

***************************************
***        MAIN PROGRAM             ***
***************************************

 OPEN PROCESS WEEKEND CID %CID AT FAC

CALL STAT_CHECK (%CID, %HRZN_OPEN, %RESULT, %REQSEND, %ADVICE)
JUMP TO (ABORT) %ADVICE - 1

*** FALL THROUGH WHEN ADVICE=1 (OPEN SUCCEEDED) ***

SEND ’HELLO, MADAME!’ TO %CID REQSEND %REQSEND

CALL STAT_CHECK (%CID, %HRZN_SEND, %RESULT, %REQSEND, %ADVICE)
JUMP TO (ABORT, -
HANDLE_ERROR, -
HANDLE_SIGNAL)   %ADVICE - 1

*** FALL THROUGH WHEN ADVICE=1 (SEND SUCCEEDED) ***

RECEIVE_LOOP:

RECEIVE %RECVMSG FROM %CID RESULT %RESULT

CALL STAT_CHECK (%CID, %HRZN_RECEIVE, %RESULT, %REQSEND,   -
  %ADVICE)
```

```
JUMP TO (ABORT, -
HANDLE_ERROR,-
BOGUS, -
SEND, -
HANDLE_CLOSE,-
HANDLE_CONFIRM)   %ADVICE - 1

*** FALL THROUGH WHEN ADVICE=1 (RECEIVED DATA) ***

 PRINT 'RECEIVED FROM PARTNER:' AND %RECVMSG

 JUMP TO RECEIVE_LOOP

SEND:

 SEND 'GOODBYE, MADAME!' TO %CID REQSEND %REQSEND

CALL STAT_CHECK (%CID, %HRZN_SEND, %RESULT, %REQSEND, %ADVICE)
JUMP TO (ABORT, -
HANDLE_ERROR, -
HANDLE_SIGNAL)   %ADVICE - 1

*** FALL THROUGH WHEN ADVICE=1 (SEND SUCCEEDED) ***

CLOSE PROCESS %CID

CALL STAT_CHECK (%CID, %HRZN_CLOSE, %RESULT, %REQSEND, %ADVICE)
JUMP TO (END, -
ABORT, -
HANDLE_ERROR)   %ADVICE

**************************************************
***         EXCEPTION HANDLERS                 ***
**************************************************

HANDLE_CONFIRM:

 PRINT '*** PARTNER REQUESTED CONFIRMATION ***'

 SEND ERROR TO %CID REQSEND %REQSEND

CALL STAT_CHECK(%CID, %HRZN_SEND_ERROR, %RESULT, %REQSEND,   -
  %ADVICE)
JUMP TO (ABORT, -
BOGUS, -
ABORT)   %ADVICE - 1

*** FALL THROUGH IF ADVICE = 1 (SEND ERROR SUCCEEDED) ***

 CLOSE PROCESS %CID

CALL STAT_CHECK (%CID, %HRZN_CLOSE, %RESULT, %REQSEND, %ADVICE)
```

```
*** IF CLOSE FAILED, JUST GIVE UP

JUMP TO (END, -
END, -
END)   %ADVICE

HANDLE_SIGNAL:

 PRINT '*** PARTNER SENT SIGNAL - CLOSING CONVERSATION ***'

                   CLOSE PROCESS %CID

CALL STAT_CHECK (%CID, %HRZN_CLOSE, %RESULT, %REQSEND, %ADVICE)

*** IF CLOSE FAILED, JUST GIVE UP

JUMP TO (END, -
END,-
END)   %ADVICE

HANDLE_ERROR:

 PRINT '*** PARTNER SENT ERROR - CLOSING CONVERSATION ***'

 CLOSE PROCESS %CID

CALL STAT_CHECK (%CID, %HRZN_CLOSE, %RESULT, %REQSEND, %ADVICE)

*** IF CLOSE FAILED, JUST GIVE UP

JUMP TO (END, -
END, -
END)   %ADVICE

HANDLE_CLOSE:

 PRINT '*** PARTNER CLOSED CONVERSATION ***'

 CLOSE PROCESS %CID

CALL STAT_CHECK (%CID, %HRZN_CLOSE, %RESULT, %REQSEND, %ADVICE)

*** IF CLOSE FAILED, JUST GIVE UP

JUMP TO (END, -
END, -
END)   %ADVICE

BOGUS:

 PRINT '*** STAT_CHECK RETURNED AN UNDEFINED ADVICE CODE ***'
 JUMP TO END
```

```
ABORT:

 PRINT '*** STAT_CHECK ADVISES THAT WE ABORT ***'
 JUMP TO END

END: END
END PROCEDURE
```

Server Sample

The following example is a server program that receives messages from a partner program and sends back a message to the partner, repeating this sequence until the partner ends the conversation.

The STAT_CHECK subroutine is used after each HORIZON statement to determine the appropriate subsequent action for the program to take. Subroutine STAT_CHECK is found in "Subroutine to Check Horizon Return Codes" on page 125.

The HORIZON LINK, PROCESSGROUP, and PROCESS definition commands are included as comments at the beginning of the procedure for convenient reference.

In this example, the server program is invoked using the Remote Procedure Invocation (RPI) subsystem. See "Application Testing with the RPI Subsystem" on page 45.

Companion program

"The following example is a small client program that establishes a conversation with a partner process, sends a message to the partner, receives messages from the partner, and ends the conversation." on page 117 has an example of a client program that can communicate with the server program shown here.

Program

```
PROCEDURE SERVER
BEGIN

*****************************************************************
***   S A M P L E  H O R I Z O N  S E R V E R  P R O G R A M   ***
*****************************************************************
******************************************
***         NETWORK DEFINITIONS        ***
******************************************
*
* DEFINE LINK BOULDER WITH SCOPE=SYSTEM LOCALID=COLORADO     -
*  INBUFSIZE=2048 TRANSPORT=VTAM PROTOCOL=LU62 SESSIONS=4
*
```

```
* DEFINE PROCESSGROUP VIRGINIA WITH SCOPE=SYSTEM OUTLIMIT=0  -
*  INLIMIT=4 REMOTEID=FREDBURG LINK=BOULDER LOGIN=NOTRUST    -
*  GUESTUSER=REJECT
*
* DEFINE PROCESS SOMEFUN WITH SCOPE=SYSTEM SUBSYSTEM=RPI     -
*  FROM=(VIRGINIA) DATALEN=500 SUBSYSPARM='MYFILE SERVER OPEN'
*
******************************************
***        DATA AND SUBROUTINES        ***
******************************************

 %CID IS STRING LEN 8
 %RECVMSG IS STRING LEN 255
 %RESULT IS STRING LEN 15
 %REQSEND IS FLOAT
 %ADVICE IS FLOAT

 %CID = 'SAILOR'

 INCLUDE STAT_CHECK

******************************************
***            MAIN PROGRAM            ***
******************************************
OPEN PROCESS SOMEFUN CID %CID ACCEPT

CALL STAT_CHECK (%CID, %HRZN_OPEN, %RESULT, %REQSEND, %ADVICE)
JUMP TO (ABORT) %ADVICE - 1

*** FALL THROUGH WHEN ADVICE=1 (OPEN SUCCEEDED) ***

AUDIT 'CONVERSATION HAS BEGUN'

RECEIVE_LOOP:

RECEIVE %RECVMSG FROM %CID RESULT %RESULT

CALL STAT_CHECK (%CID, %HRZN_RECEIVE, %RESULT, %REQSEND,  -
  %ADVICE)
JUMP TO (ABORT, -
HANDLE_ERROR, -
BOGUS, -
SEND, -
HANDLE_CLOSE, -
HANDLE_CONFIRM) %ADVICE - 1

*** FALL THROUGH WHEN ADVICE=1 (RECEIVED DATA) ***

AUDIT 'RECEIVED FROM PARTNER:' AND %RECVMSG

JUMP TO RECEIVE_LOOP
```

```
SEND:

SEND ’HELLO, SAILOR!’ TO %CID REQSEND %REQSEND

CALL STAT_CHECK (%CID, %HRZN_SEND, %RESULT, %REQSEND, %ADVICE)
JUMP TO (ABORT, -
HANDLE_ERROR, -
HANDLE_SIGNAL) %ADVICE - 1

*** FALL THROUGH WHEN ADVICE=1 (SEND SUCCEEDED) ***

JUMP TO RECEIVE_LOOP

******************************************
***         EXCEPTION HANDLERS         ***
******************************************

HANDLE_CONFIRM:

AUDIT ’*** PARTNER REQUESTED CONFIRMATION ***’

SEND ERROR TO %CID REQSEND %REQSEND

CALL STAT_CHECK(%CID, %HRZN_SEND_ERROR, %RESULT, %REQSEND,  -
 %ADVICE)
JUMP TO (ABORT, -
BOGUS, -
ABORT)  %ADVICE - 1

*** FALL THROUGH IF ADVICE = 1 (SEND ERROR SUCCEEDED) ***

CLOSE PROCESS %CID

*** IF CLOSE FAILS, JUST GIVE UP

CALL STAT_CHECK (%CID, %HRZN_CLOSE, %RESULT, %REQSEND, %ADVICE)
JUMP TO (END, -
END, -
END) %ADVICE

HANDLE_SIGNAL:

AUDIT ’*** PARTNER SENT SIGNAL - CLOSING CONVERSATION ***’

CLOSE PROCESS %CID

*** IF CLOSE FAILS, JUST GIVE UP

CALL STAT_CHECK (%CID, %HRZN_CLOSE, %RESULT, %REQSEND, %ADVICE)
JUMP TO (END, -
END, -
END)  %ADVICE

HANDLE_ERROR:

AUDIT ’*** PARTNER SENT ERROR - CLOSING CONVERSATION ***’

CLOSE PROCESS %CID

*** IF CLOSE FAILS, JUST GIVE UP

CALL STAT_CHECK (%CID, %HRZN_CLOSE, %RESULT, %REQSEND, %ADVICE)
JUMP TO (END, -
END, -
END)  %ADVICE

HANDLE_CLOSE:

AUDIT ’*** PARTNER CLOSED CONVERSATION ***’

CLOSE PROCESS %CID

*** IF CLOSE FAILS, JUST GIVE UP

CALL STAT_CHECK (%CID, %HRZN_CLOSE, %RESULT, %REQSEND, %ADVICE)
JUMP TO (END, -
END, -
END)  %ADVICE



BOGUS:

AUDIT ’*** STAT_CHECK RETURNED AN UNDEFINED ADVICE CODE ***’
JUMP TO END

ABORT:

AUDIT ’*** STAT_CHECK ADVISES THAT WE ABORT ***’
JUMP TO END

END:

*** ALWAYS SET THE APSY COMMUNICATION VARIABLE BEFORE LEAVING

%X = $SETG(’COMM’,’EXIT’)
END
END PROCEDURE
```

Subroutine to Check Horizon Return Codes

The following subroutine can be used in Horizon User Language (SOUL) programs to simplify the checking of the status information that is set by each HORIZON statement. The subroutine condenses the large number of combinations of $STATUS codes, RESULT codes, and so on into a small set of recommended actions for the program:

- Continue normally.
- Terminate immediately.
- Respond to partner’s SEND ERROR.
- Respond to partner’s SIGNAL PROCESS.
- Start to SEND.
- Partner has closed conversation.
- Respond to partner’s CONFIRM.
- No more data expected.
- Expected data has not arrived.

Refer to the client and server program samples in this chapter on page 117 and page 121, respectively, for illustrations of how the subroutine is to be used.

Subroutine

```
PROCEDURE STAT_CHECK
***********************************************************************
***

SUBROUTINE STAT_CHECK (%PROCESS IS STRING LEN 10 INPUT-
%STATEMENT IS FLOAT INPUT, -
%RESULT IS STRING LEN 15 INPUT, -
%REQSEND IS FLOAT INPUT, -
%ADVICE IS FLOAT OUTPUT )

***********************************************************************
* STAT_CHECK SHOULD BE CALLED AFTER EACH HORIZON STATEMENT.           *
* IT EVALUATES THE RELEVANT STATUS INFORMATION THAT WAS SET BY        *
* THE STATEMENT AND RETURNS A RECOMMENDED PROGRAM-ACTION CODE.        *
*                                                                     *
* %PROCESS  -- THE CONVERSATION ID (CID) OF THE PROCESS               *
* %STATEMENT-- THE HORIZON STATEMENT JUST ISSUED                      *
*   1 = OPEN (O)          8 = SEND (S)                                *
*   2 = CONFIRM (CF)      9 = SEND ERROR (SE)                         *
*   3 = CONFIRMED (CFD)  10 = SIGNAL (SP)                             *
*   4 = CLOSE (C)        11 = INVITE (I)                              *
*   5 = FLUSH (F)        12 = TEST (T)                               *
*   6 = QUERY (Q)        13 = WAIT (W)                               *
*   7 = RECEIVE (R)                                                  *
* %RESULT   -- THE RESULT VARIABLE FROM THE 'RECEIVE' STATEMENT      *
* %REQSEND  -- THE REQSEND VARIABLE FROM 'SEND' OR 'SEND ERROR'      *
* %ADVICE   -- (OUTPUT) THE RECOMMENDED ACTION TO TAKE.  USE THE     *
*     FOLLOWING CHART TO DETERMINE WHICH ADVICE CODES                *
*     STAT_CHECK CAN RETURN FOR DIFFERENT HORIZON                    *
*     STATEMENTS:                                                    *
*                                                                    *
* - - - S T A T E M E N T S - - -    - - A D V I C E  C O D E S - -  *
*                                                                    *
* O CF CFD C F I Q R S SE SP T W                                     *
* X X  X   X X X X X X  X  X X X    1 - CONTINUE                     *
* X X  X   X X X X X X  X  X X X    2 - ABORT                        *
* - X  -   X - X - X X  -  - - -    3 - GOT 'SEND ERROR'             *
* - X  -   - - - - - X  X  - - -    4 - GOT 'SIGNAL PROCESS'         *
* - -  X   - - - - X -  -  - - -    5 - TRANSITION TO 'SEND'         *
* - -  X   - - X - X -  -  - - -    6 - PARTNER CLOSED CONVERSATION  *
* - -  -   - - - - X -  -  - - -    7 - TRANSITION TO 'CONFIRM'      *
* - -  -   - - - - - -  -  - X X    8 - NO OUTSTANDING INVITES       *
* - -  -   - - - - - -  -  - X -    9 - NO RECEIPTS YET              *
**********************************************************************

%STATUS = $STATUS
%STATUSD = $STATUSD

JUMP TO (O,CF,CFD,C,F,Q,R,S,SE,SP,I,T,W) %STATEMENT

X:  **********************************
    *      INVALID STATEMENT CODE    *
    **********************************

    JUMP TO ABORT

O:  **********************************
    *          OPEN PROCESS          *
    **********************************

    IF %STATUS = 0 THEN
    JUMP TO CONTINUE
    ELSE
    JUMP TO ABORT
    END IF

CF: **********************************
    *            CONFIRM             *
    **********************************

    *** FIRST SEE IF SIGNAL RECEIVED FROM PARTNER
```

```
 IF %REQSEND = 1 THEN
  JUMP TO REQUEST_TO_SEND
 ELSEIF %STATUS = 0 THEN
  JUMP TO CONTINUE

  *** CHECK FOR 2/2 (PARTNER RESPONDED TO CONFIRM WITH SEND ERROR)

 ELSEIF %STATUS = 2 AND %STATUSD = 2 THEN
  JUMP TO SEND_ERROR
 ELSE
  JUMP TO ABORT
 END IF

CFD: **********************************
     *           CONFIRMED            *
     **********************************

 %S IS STRING LEN 8

 IF %STATUS = 0 THEN



*** SEE WHAT THE CONVERSATION STATE HAS CHANGED TO

  QUERY PROCESS %PROCESS STATE %S
  IF %S = 'CLOSE' THEN
   JUMP TO CLOSE_BY_REMOTE
  ELSEIF %S = 'SEND' THEN
   JUMP TO SEND
  ELSE
   JUMP TO CONTINUE
  END IF

 ELSE
  JUMP TO ABORT
 END IF

C:  **********************************
    *         CLOSE PROCESS          *
    **********************************

 IF %STATUS = 0 THEN
  JUMP TO CONTINUE

     *** CHECK FOR 2/2 (PARTNER RESPONDED TO CONFIRM WITH SEND ERROR)

 ELSEIF %STATUS = 2 AND %STATUSD = 2 THEN
  JUMP TO SEND_ERROR
 ELSE
```

```
  JUMP TO ABORT
 END IF

F:  **********************************
    *         FLUSH PROCESS          *
    **********************************

 IF %STATUS = 0 THEN
  JUMP TO CONTINUE
 ELSE
  JUMP TO ABORT
 END IF

Q:  **********************************
    *         QUERY PROCESS          *
    **********************************

 IF %STATUS = 0 THEN
  JUMP TO CONTINUE
 ELSE
  JUMP TO ABORT
 END IF

R:  **********************************
    *            RECEIVE             *
    **********************************

 IF %STATUS = 0 THEN
  JUMP TO CONTINUE
 ELSEIF %STATUS = 1 THEN

  *** ANALYZE THE RESULT VARIABLE FOR APPROPRIATE ACTION

  *** DATA TRUNCATED OR INCOMPLETE - ASSUME THIS ISN’T DESIRABLE

  IF %RESULT = ’DATA TRUNCATED’ THEN
   JUMP TO ABORT
  ELSEIF %RESULT = ’DATA INCOMPLETE’ THEN
   JUMP TO ABORT
  ELSEIF %RESULT = ’SEND’ THEN
   JUMP TO SEND

   *** ALL FLAVORS OF CONFIRM STATE - TELL USER TO DO ’CONFIRMED’.
    *** AFTERWARD, STAT_CHECK WILL SAY ’SEND’, ’CLOSE’, ETC.

  ELSEIF %RESULT = ’CONFIRM’ THEN
   JUMP TO CONFIRM
  ELSEIF %RESULT = ’CONFIRM SEND’ THEN
   JUMP TO CONFIRM
  ELSEIF %RESULT = ’CONFIRM CLOSE’ THEN
```

```
   JUMP TO CONFIRM
  ELSE
   JUMP TO ABORT
  END IF

   *** CHECK FOR 2/2 (PARTNER RESPONDED TO CONFIRM WITH SEND ERROR)

 ELSEIF %STATUS = 2 AND %STATUSD = 2 THEN
  JUMP TO SEND_ERROR

   *** CHECK FOR 4/0 (INDICATES PARTNER ENDED CONVERSATION NORMALLY)

 ELSEIF %STATUS = 4 AND %STATUSD = 0 THEN
  JUMP TO CLOSE_BY_REMOTE
 ELSE
  JUMP TO ABORT
 END IF

S:  **********************************
    *              SEND              *
    **********************************

 IF %STATUS = 0 THEN

    *** SEE IF SIGNAL RECEIVED FROM PARTNER

  IF %REQSEND = 1 THEN
   JUMP TO REQUEST_TO_SEND
  ELSE
   JUMP TO CONTINUE
  END IF

   *** CHECK FOR 2/2 (PARTNER RESPONDED TO CONFIRM WITH SEND ERROR)

 ELSEIF %STATUS = 2 AND %STATUSD = 2 THEN
  JUMP TO SEND_ERROR
 ELSE
  JUMP TO ABORT
 END IF

SE: **********************************
    *          SEND ERROR            *
    **********************************

   *** SEE IF SIGNAL RECEIVED FROM PARTNER

 IF %REQSEND = 1 THEN
  JUMP TO REQUEST_TO_SEND

   *** NO SIGNAL, SO CHECK %STATUS
 ELSEIF %STATUS = 0 THEN
  JUMP TO CONTINUE
 ELSE
  JUMP TO ABORT
 END IF

SP: **********************************
    *         SIGNAL PROCESS         *
    **********************************

 IF %STATUS = 0 THEN
  JUMP TO CONTINUE
 ELSE
  JUMP TO ABORT
 END IF

I:  **********************************
    *         INVITE                 *
    **********************************

 IF %STATUS = 0 THEN
  JUMP TO CONTINUE
 ELSEIF %STATUS = 2 AND %STATUSD = 2 THEN
  JUMP TO SEND_ERROR
 ELSEIF %STATUS = 4 THEN
  IF %STATUSD = 0 THEN
   JUMP TO CLOSE_BY_REMOTE
  ELSE
   JUMP TO ABORT
  END IF
 ELSE
  JUMP TO ABORT
 END IF
 T: **********************************
    *       TEST FOR RECEIPT         *
    **********************************

 IF %STATUS = 0 THEN
  JUMP TO CONTINUE
 ELSEIF %STATUS = 1 THEN
  IF %STATUSD = 1 THEN
   JUMP TO NEVER_MORE
  ELSEIF %STATUSD = 2 THEN
   JUMP TO NOT_YET
  ELSE
   JUMP TO ABORT
  END IF
 ELSE
  JUMP TO ABORT
 END IF
 W: **********************************
    *       WAIT FOR RECEIPT         *
    **********************************

 IF %STATUS = 0 THEN
  JUMP TO CONTINUE
 ELSEIF %STATUS = 1 THEN
  IF %STATUSD = 2 THEN
   JUMP TO NEVER_MORE
  END IF
 END IF
 JUMP TO ABORT

******************************************************************
***              SET THE ADVICE CODE AND EXIT.                 ***
******************************************************************
CONTINUE:
 %ADVICE = 1
 JUMP TO EXIT

ABORT:
 %ADVICE = 2
 JUMP TO EXIT

SEND_ERROR:
 %ADVICE = 3
 JUMP TO EXIT

REQUEST_TO_SEND:
 %ADVICE = 4
 JUMP TO EXIT

SEND:
 %ADVICE = 5
 JUMP TO EXIT

CLOSE_BY_REMOTE:
 %ADVICE = 6
 JUMP TO EXIT

CONFIRM:
 %ADVICE = 7
 JUMP TO EXIT

NEVER_MORE:
 %ADVICE = 8
 JUMP TO EXIT

NOT_YET:
 %ADVICE = 9
 JUMP TO EXIT

*******************************************************************
***                   SUBROUTINE EXIT PATH                      ***
*******************************************************************


EXIT:

*******************************************************************
***  NOTE: THE FOLLOWING AUDIT LOGIC IS USEFUL AS A DEBUGGING  ***
***  TOOL FOR TRACING HORIZON STATEMENTS BUT MAY BE REMOVED    ***
***  OR BYPASSED FOR EFFICIENCY IF DESIRED.                    ***
*******************************************************************
   *** AUDIT RESULT OF EACH STAT_CHECK INVOCATION ***

 %STMT IS STRING LEN 14

 IF %STATEMENT = 1 THEN
  %STMT = 'OPEN'
 ELSEIF %STATEMENT = 2 THEN
  %STMT = 'CONFIRM'
 ELSEIF %STATEMENT = 3 THEN
  %STMT = 'CONFIRMED'
 ELSEIF %STATEMENT = 4 THEN
  %STMT = 'CLOSE'
 ELSEIF %STATEMENT = 5 THEN
  %STMT = 'FLUSH'
 ELSEIF %STATEMENT = 6 THEN
  %STMT = 'QUERY'
 ELSEIF %STATEMENT = 7 THEN
  %STMT = 'RECEIVE'
 ELSEIF %STATEMENT = 8 THEN
  %STMT = 'SEND'
 ELSEIF %STATEMENT = 9 THEN
  %STMT = 'SEND ERROR'
 ELSEIF %STATEMENT = 10 THEN
  %STMT = 'SIGNAL PROCESS'
 ELSEIF %STATEMENT = 11 THEN
  %STMT = 'INVITE'
 ELSEIF %STATEMENT = 12 THEN
  %STMT = 'TEST'
 ELSEIF %STATEMENT = 13 THEN
  %STMT = 'WAIT'
 ELSE
  %STMT = '**UNDEFINED**'
 END IF

 %ADV IS STRING LEN 20
```

```
IF %ADVICE = 1 THEN
 %ADV = ’CONTINUE’

ELSEIF %ADVICE = 2 THEN
 %ADV = ’ABORT’
ELSEIF %ADVICE = 3 THEN
 %ADV = ’HANDLE SEND ERROR’
ELSEIF %ADVICE = 4 THEN
 %ADV = ’HANDLE SIGNAL’
ELSEIF %ADVICE = 5 THEN
 %ADV = ’START SENDING’
ELSEIF %ADVICE = 6 THEN
 %ADV = ’PARTNER CLOSED’
ELSEIF %ADVICE = 7 THEN
 %ADV = ’HANDLE CONFIRM’
ELSEIF %ADVICE = 8 THEN
 %ADV = ’NOTHING ACTIVE’
ELSEIF %ADVICE = 9 THEN
 %ADV = ’NOTHING YET’
END IF

QUERY PROCESS %PROCESS STATE %S

%TRACE IS STRING LEN 255
%TRACE = ’STAT_CHECK:’           -
 WITH ’ Statement=’ WITH %STMT    -
 WITH ’ %STATUS=’ WITH %STATUS WITH ’/’ WITH %STATUSD

   *** AUDIT RESULT VARIABLE AFTER RECEIVE

IF %STATEMENT = 7 THEN
 %TRACE = %TRACE WITH ’ Result=’ WITH %RESULT

  *** AUDIT REQSEND VARIABLE AFTER SEND OR SEND ERROR

ELSEIF %STATEMENT = 8 OR %STATEMENT = 9 THEN
 %TRACE = %TRACE WITH ’ Reqsend=’ WITH %REQSEND
END IF

%TRACE = %TRACE                                  -
 WITH ’ State=’ WITH %S                          -
 WITH ’ Advice=’ WITH %ADVICE WITH ’/’ WITH %ADV  -
 WITH ’ Cid=’ WITH %PROCESS

 *** USE ’PRINT’ FOR CLIENT PROGRAM, ’AUDIT’ FOR SERVER PROGRAM

IF $VIEW(’IODEV’) EQ ’27’ THEN
 AUDIT %TRACE
ELSE
```

```
  PRINT %TRACE
 END IF
******************************************************************
***                      END OF AUDITING                      ***
******************************************************************

RETURN: RETURN
END SUBROUTINE STAT_CHECK

*******************************************************************
*                                                                 *
* THE FOLLOWING VARIABLES MAY BE USED AS THE %STATEMENT PARAMETER   *
* IN A CALL TO STAT_CHECK, TO MAKE YOUR PROGRAM EASIER TO READ    *
* (BE SURE TO EXECUTE THESE ASSIGNMENT STATEMENTS *BEFORE* YOU    *
* REFERENCE THE VARIABLES.)                                       *
*                                                                 *
* EXAMPLE: OPEN PROCESS %CID                                      *
* CALL STAT_CHECK(%CID,%HRZN_OPEN,%RESULT,%REQSEND,%ADVICE)       *
*                                                                 *
*******************************************************************
%HRZN_OPEN          = 1
%HRZN_CONFIRM       = 2
%HRZN_CONFIRMED     = 3
%HRZN_CLOSE         = 4
%HRZN_FLUSH         = 5
%HRZN_QUERY         = 6
%HRZN_RECEIVE       = 7
%HRZN_SEND          = 8
%HRZN_SEND_ERROR    = 9
%HRZN_SIGNAL        = 10
%HRZN_INVITE        = 11
%HRZN_TEST          = 12
%HRZN_WAIT          = 13
END PROCEDURE
```

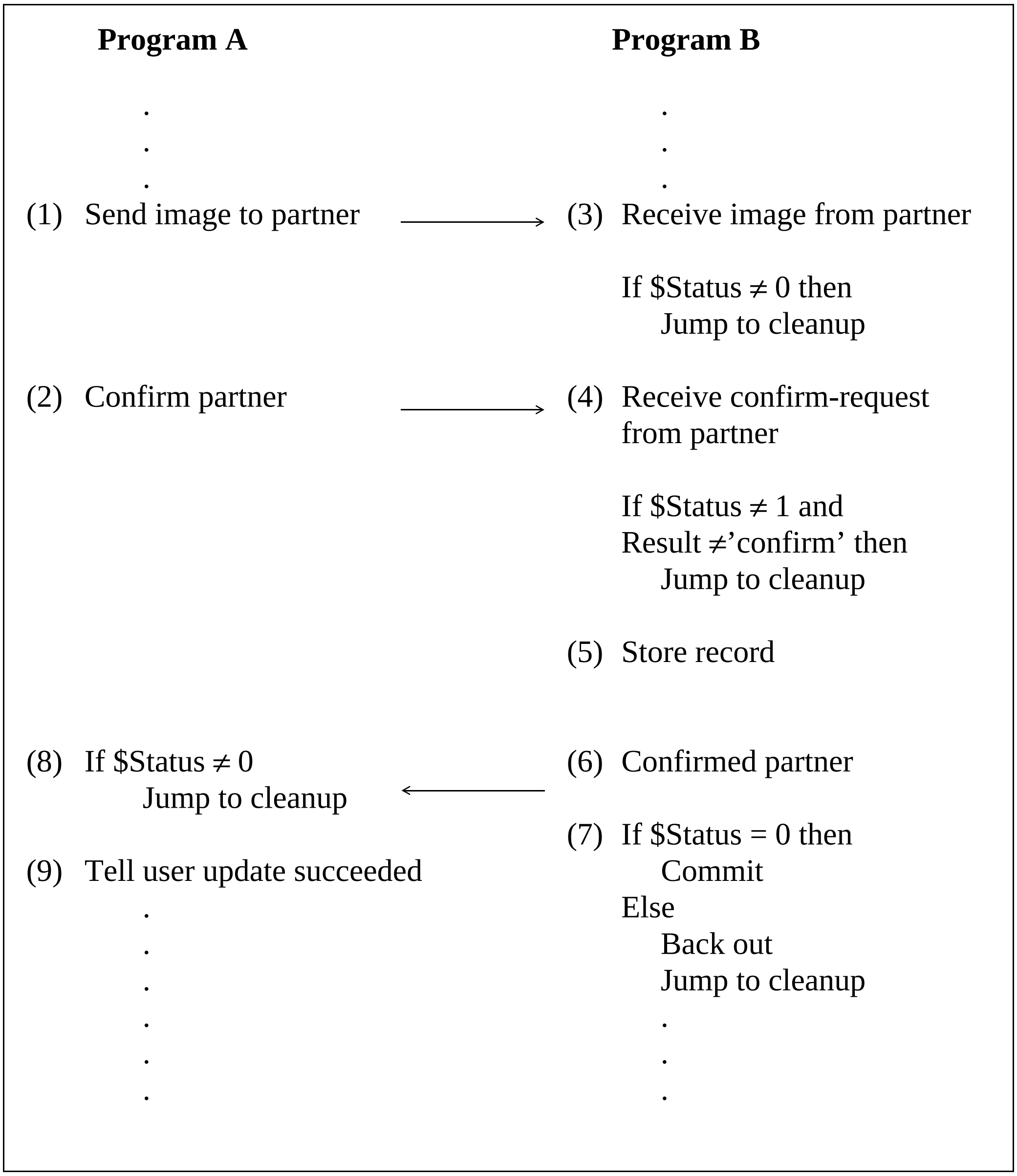
Remote Updating Example

The applications considered thus far have been inquiry-only, but Horizon also permits remote file updating. By definition, remote updating of a local node occurs when data received at the local node during a Horizon conversation is stored at the local node. Horizon accommodates this single node updating using standard Model 204 recovery features.

Coordinated updating and recovery between multiple nodes is not provided with Version 2, Release 2 of Horizon. Multiple-node updating and recovery requires an additional conversation protocol, called two-phase commit, along with corresponding enhancements to the recovery logic of Model 204.

The example (in pseudo code) in Figure 7-1 on page 135 illustrates how a Horizon single-node remote update can be accomplished using existing

Model 204 recovery logic. The arrows in the example indicate when data is physically transmitted across the network. The numbers in parentheses show the order in which the statements in the programs are processed.



```
     Program A                                  Program B

        .                                          .
        .                                          .
        .                                          .
(1)  Send image to partner      ---------->  (3)  Receive image from partner

                                                  If $Status ≠ 0 then
                                                     Jump to cleanup

(2)  Confirm partner            ---------->  (4)  Receive confirm-request
                                                  from partner

                                                  If $Status ≠ 1 and
                                                  Result ≠'confirm' then
                                                     Jump to cleanup

                                             (5)  Store record


(8)  If $Status ≠ 0                          (6)  Confirmed partner
        Jump to cleanup         <----------
                                             (7)  If $Status = 0 then
(9)  Tell user update succeeded                      Commit
        .                                         Else
        .                                            Back out
        .                                            Jump to cleanup
        .                                          .
        .                                          .
        .                                          .
```

Figure 7-1. Single-Node Remote Update

A brief description follows of the steps in parentheses in the example in Figure 7-1 on page 135*:*

Step	Description
(1)	Program A sends data to its remote partner.
(2)	Program A issues a confirmation request. This causes the data and a confirmation indicator to be sent to Program B.
(3)	Program B receives the data. If the receive fails for any reason, the appropriate cleanup logic is invoked.
(4)	Program B does a second RECEIVE, and verifies that what it has received is the confirmation request. If the confirmation request is not present, the cleanup logic is called. An update unit is never started.
(5)	Program B stores the data it receives into its file, and this begins an update unit.
(6)	Program B acknowledges that it has made the update by issuing a CONFIRMED statement.
(7)	If Program A is still active, $STATUS from the confirmation is 0, and Program B commits the current update unit. Otherwise, Program B backs the update out.
(8-9)	When something (a confirmation or an error indication) is received, Program A reports the result to the end user.

Multiple-node updating

An application that requires updates at more than one node can be written using the current version of Horizon, provided that it does not depend upon Model 204 recovery alone to assure it of database consistency across network nodes. Such an application can be designed in one of two ways:

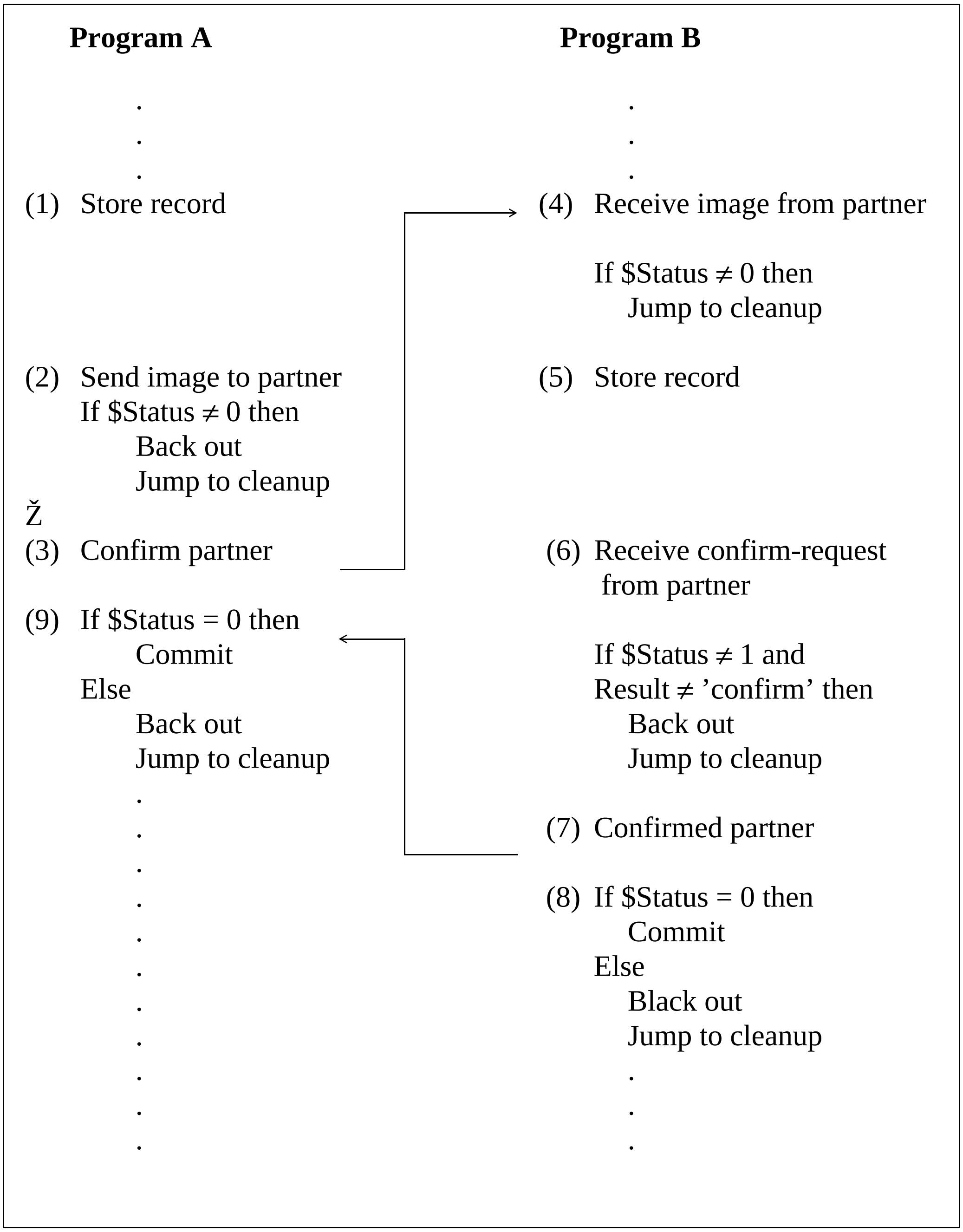
1. The application is designed so that there are no database consistency dependencies across nodes.

   In this case, Model 204 recovery protects each individual node. Such a design implies that each node’s update is seen by recovery as a separate transaction, disjoint from updates at any other node.

2. The application is designed so that the application takes responsibility for detecting and correcting cross-node inconsistencies.

   Such a recovery scheme can sometimes be designed, but an extensive discussion of the issues involved is beyond the scope of this guide.

Figure 7-Figure 7-2. shows an application that performs updates at two nodes. The numbers in parentheses show the order in which the statements in the programs are processed.



Program A

```
        .
        .
        .
(1)  Store record


(2)  Send image to partner
     If $Status ≠ 0 then
          Back out
          Jump to cleanup
Ž
(3)  Confirm partner

(9)  If $Status = 0 then
          Commit
     Else
          Back out
          Jump to cleanup
          .
          .
          .
          .
          .
          .
          .
          .
          .
          .
          .
```

Program B

```
        .
        .
        .
(4)  Receive image from partner

     If $Status ≠ 0 then
        Jump to cleanup

(5)  Store record


(6)  Receive confirm-request
      from partner

     If $Status ≠ 1 and
     Result ≠ 'confirm' then
        Back out
        Jump to cleanup

(7)  Confirmed partner

(8)  If $Status = 0 then
        Commit
     Else
        Black out
        Jump to cleanup
        .
        .
        .
```

Figure 7-2. Double-Node Update

A brief description follows the steps in the example in Figure 7-2 on page 137:

Steps	Description
(1)	Program A stores data into a record, and this begins an update unit.
(2)	Program A sends a copy of the data to its remote partner.
(3)	Program A issues a confirmation request. This causes the data and a confirmation indicator to be sent to Program B.
(4)	Program B receives the data. If the RECEIVE fails for any reason, the appropriate cleanup logic is invoked.
(5)	Program B stores the data it receives into its file. This begins an update unit.
(6)	Program B verifies that what it has received is the confirmation request. If the confirmation request is not present, the update unit is backed out and the cleanup logic is called.
(7)	Program B acknowledges that it has made the update by issuing a CONFIRMED statement.
(8)	This step begins when the confirmation is sent. If Program A is still active, $STATUS from the confirmation is 0, and Program B commits the current update unit.
(9)	When something (a confirmation or an error indication) is received, Program A acts according to what it receives: • A confirmation ($STATUS=0) signifies that Program B successfully received the update data and stored its record, and is presently committing the update. Program A can therefore commit its update with the near certainty that its partner's commit will also succeed. • An error ($STATUS¼0) signifies that, for some reason, the Program B update did not succeed. Program A therefore backs out its update.

Between-node data consistency

A database consistency problem can occur if Program A terminates before it gets a chance to make the decision at Step 9, or if Program B terminates before it completes Step 8. The application should be designed to check (perhaps at initialization time or just before performing a given update) that the database at each node "looks correct."

Another alternative is for the initiating node to keep a file to which it logs records of the nodes that have been successfully updated. In the event of a network failure, the data in this log file is used for manually coordinating recovery.

Note: The physical consistency of any Model 204 file is always taken care of by Model 204 recovery, and is not a consideration here. Logical consistency within a single Model 204 system is also provided. The only issue that the application designer needs to worry about is logical consistency of data among the different nodes that the application attempts to update.

8

Horizon Error Processing

Overview of Horizon error processing

Most Horizon errors do not cause the SOUL program to be cancelled. Usually, however, the appropriate programmatic action when an error is detected is to display diagnostic information, terminate the program, and debug the problem.

Debugging program-to-program errors is more difficult than debugging a single program, because there are more places where problems may occur. There are two or more programs, running asynchronously at different nodes, and there are numerous network components connecting them.

This chapter discusses the detection and troubleshooting of typical types of Horizon error conditions and provides a table containing the Horizon error codes and messages.

The Rocket Model 204 messages documentation contains descriptive and debugging information for many of the Horizon error conditions that are accompanied by a Model 204 error message number.

Detecting Horizon Errors

Unlike most other SOUL statements, when a Horizon statement is executed, certain status information is set to indicate whether the statement completed normally. In cases of error, this status information may not only indicate the type of error but also may contain directions about how to correct the error.

It is the responsibility of the application program to check this status information and to detect and handle errors.

Conversation status information

Information about the current condition of a Horizon conversation is communicated to the SOUL program through the following means:

- The $STATUS and $STATUSD functions.
- The RESULT variable (on RECEIVE statements).
- The REQSEND variable (on the SEND, CONFIRM, and SEND ERROR statements).
- The QUERY PROCESS statement, for additional information, such as conversation state.

For more about the RESULT, REQSEND, and conversation state information pertaining to individual Horizon conversation interface statements, see the descriptions of each statement in the Chapter 6.

The execution of a Horizon SOUL statement always sets $STATUS to some integer. Since each $STATUS code provides general information about the current condition of the conversation, the application program should include some logic to test $STATUS after each Horizon statement. When $STATUS indicates an error has occurred ($STATUS=2 or greater), a Model 204 message is generated.

The individual $STATUS (and $STATUSD) codes and messages are listed in tabular format in “$STATUS/$STATUSD Codes and Accompanying Error Messages” on page 144.

Note: Model 204 (as opposed to Horizon) error messages generated during Horizon program execution are not written to the user’s terminal. They can be retrieved programmatically, however, using the $ERRMSG function.

Client versus server error detection and debugging

Client processes usually run on a thread that has a terminal. The exceptional situation is when a server process initiates a conversation (becomes a client process) with a third partner. The presence of the terminal permits interactive debugging: the developer can insert PRINT statements into trace processing routines to display status information about the conversation and to create break points, for example.

A server process, on the other hand, does not have access to a terminal. Messages are only written to the audit trail or the operator’s console. Debugging a server process usually involves embedding AUDIT (rather than PRINT) statements and browsing the audit trail for messages originating from the thread on which the server process ran.

Troubleshooting Horizon Errors

In this section, the general categories of Horizon errors are listed, followed by discussions of typical debugging approaches for OPEN PROCESS errors and errors producing unanticipated conversation terminations.

Types of errors

There are several types of Horizon errors, corresponding to different values of $STATUS. The list below describes briefly the types of errors. All the $STATUS (and $STATUSD) codes and messages are listed in tabular format starting on page 144.

- State check ($STATUS=3).

  The program has issued a Horizon SOUL statement that is invalid for the current state of the conversation. State checks are described further in “Horizon Conversation Data Flow” on page 79.

- Partner closed the conversation ($STATUS=4).

  This occurs if a partner has terminated or issued a CLOSE PROCESS statement.

- Parameter check ($STATUS=5).

  One or more parameters of a Horizon statement or Horizon entity definition is incorrect.

- System failures ($STATUS=10 or greater).

  A Horizon system component has failed or is unavailable. Unlike the other types of errors, this is generally not an application program bug. There are four kinds of system failures:

  – Local allocation: a conversation could not be established because of a lack of resources at the node where the program is running.

  – Remote allocation: a conversation could not be established because of a lack of resources at the node to which the connection was being attempted.

  – Session: failure of the LU 6.2 session on which the conversation was running.

  – Link: a SNA Communications Server (formerly VTAM) error occurred.

- Partner issued SEND ERROR statement ($STATUS=2).

  Unlike the other types of errors, this is an application-level error. It allows a program to create an error condition that is communicated to its partner. For more information about the SEND ERROR statement, see page 111.

OPEN PROCESS errors

While many of the types of errors described in the previous section can occur throughout the life of a conversation, one error situation is likely to be the most frequent: a system failure encountered during a client program’s OPEN PROCESS statement and the subsequent attempt at the server node to start the requested partner process.

As described on page 103, such an error is not always received immediately following the OPEN PROCESS statement. Most Horizon statements simply cause LU 6.2 requests to be buffered until physical transmission becomes necessary, which may be several Horizon statements later. Consequently, a client program may not receive an indication of a remote allocation failure until some later statement.

For more information about the physical sending of requests, see “Buffering and Shipping Conversation Information” on page 88.

OPEN PROCESS errors received immediately

The types of errors typically seen at OPEN PROCESS time on the client side are the following:

- State Check
- Parameter Check
- Retryable and Non-Retryable Local Allocation errors

Generally the remedy is simple. For example, suppose that following an OPEN PROCESS statement, $STATUS=5 and $STATUSD=4. This is a parameter check error: entity not defined. After making sure that the process name is correctly spelled, check with the system manager to make sure a process definition exists.

When an error from the server side is returned on a subsequent Horizon statement, tracking it down is not so simple.

OPEN PROCESS errors traced to the server

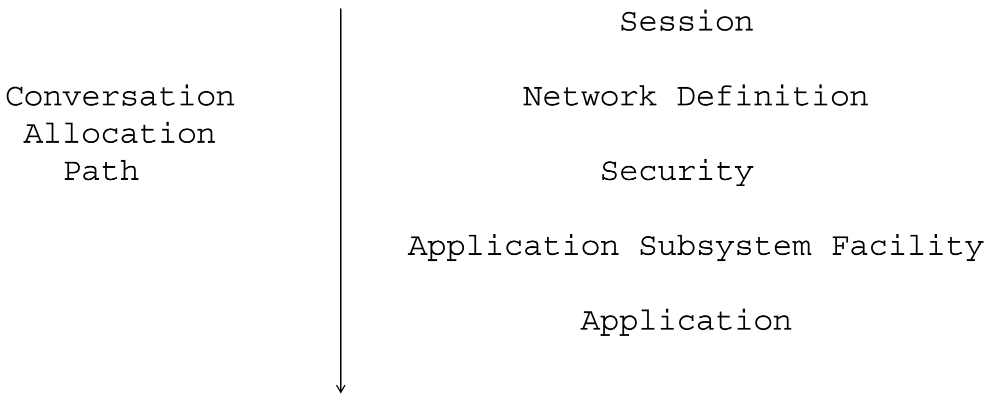
When a client program receives an error that can be traced to the server side, it is almost always necessary to look in the server system’s audit trail for an indication of what happened. Because Horizon is based on the LU 6.2 protocol, only limited information can be transmitted along with an error: The return of detailed information about why the conversation failed is considered a breach of security.

For example, a server program may fail due to a shortage of any one of a number of Model 204 or network resources. A Model 204 error message pinpointing the reason for the failure is generated at that server node; however, the error notification received by the client program only indicates that a resource shortage occurred at the server. The audit trail at the server node must be looked at to find out what resource is not available.

For security violations, Horizon sends only an indication that there has been a security violation. To determine if the USERID or PASSWORD value is incorrect, for example, it is necessary to look in the audit trail at the server node.

Troubleshooting conversation initiation layers for server errors

To determine where the fault is on the server side, consider the layers that a conversation request must pass through for the conversation to start:



To determine from the client side where the server problem is, the developer should examine each layer in turn. Examples follow of kinds of errors at each layer:

- If there are no sessions available or the remote LU is not active, the conversation fails.
- If the Horizon network definitions are not correct, the remote LU may not be known at the local node.
- If the security requirements at the remote node are not met, the conversation fails.
- If the server program is not defined, the conversation fails.
- If the partner application subsystem is not started, the conversation fails.

Since each layer can generate a variety of errors, a systematic examination of the conversation initiation layers is the most productive debugging approach.

Unanticipated conversation termination

Another error, which can occur at any time, is an abnormal termination of conversation by the partner ($STATUS=4, $STATUSD=1). A variety of events at the partner node can result in abnormal conversation termination, including:

- Bumping of the partner thread.
- Termination of the partner process due to program errors.
- Issuing of CLOSE LINK FORCE by the partner's LU.

The only way to determine what went wrong is to examine the audit trail at the partner node for messages associated with the conversation that ended.

$STATUS/$STATUSD Codes and Accompanying Error Messages

Table 8-1 lists the possible combinations of return values of the $STATUS and $STATUSD functions (S/SD) and a brief description of each combination.

In addition to the $STATUS and $STATUSD information, you can use $ERRMSG to retrieve the Model 204 error message numbers generated along with $STATUS and $STATUSD. The message text associated with the message number (along with debugging hints when appropriate) is found in the Rocket Model 204 messages documentation.

Table 8-1. $STATUS/D Codes and Error Message Numbers

S	SD	Description	EMN
0	0	Normal completion; no unusual conditions.	none
1	0	Normal completion; check RESULT variable.	none
1	1	Specified event(s) not pending (WAIT or TEST).	none
1	2	Specified event(s0 still pending (TEST).	none
1	3	WAIT completed due to timeout.	none
2	2	SEND ERROR statement issued by partner (Horizon only)	none
3	3	State check: verb issued in wrong conversation state.	1254
4	0	Conversation ended normally by partner process. The partner process has finished executing, or it has ended the conversation by issuing a CLOSE PROCESS statement.	none
4	1	Partner process closed conversation abnormally.	none
5	1	Password required. If the USERID keyword parameter is provided on the OPEN PROCESS statement, the PASSWORD parameter must also be provided.	1950
5	2	Process is already open. The OPEN PROCESS statement being executed specifies a process name or CID for a process that is already open.	1499
5	3	Initial (PIP) data incorrectly specified.	2031
5	4	Entity (process, processgroup, or link) not defined.	1743

Table 8-1. $STATUS/D Codes and Error Message Numbers (Continued)

S	SD	Description	EMN
5	5	Process has not been opened.	1572
5	6	Statement or option not supported.	1811
5	11	Process may only be opened within an application subsystem.	1948
5	12	Process definition requires ’OPEN’ for security parameters. The OPEN PROCES STATEMENT being executed contains an ACCOUNT or USERID parameter. This is only valid when the process definition specifies ACCTSOURCE=OPEN or UIDSOURCE=OPEN (PROFILE and PROFSOURCE may be written instead of ACCOUNT and ACCTSOURCE).	1949
5	13	Security violation.	1951* 2027**
5	14	SESPARMS was specified on the processgroup definition but is not valid for Horizon links.	none
5	15	OPEN type conflicts with PROCESS type.	1953* 2027**
5	16	Reserved names may not be used in OPEN PROCESS.	1955
5	17	Process name or CID is too long.	2035
5	18	Synchronization level not supported. The process issued the CONFIRM verb but the process definition does not contain the CONFIRM parameter	2036
5	19	Parameter required by not specified.	1797
5	20	Invalid duration value specified for WAIT statement.	1882
10	1	Insufficient main storage available at a local node.	1788
10	2	Reserved.	
10	3	Local link is not open or is being closed.	1494
10	4	Local Processgroup is being stopped.	1493
11	1	Partner process temporarily cannot be started.	2031
11	3	Retryable remote allocation failure.	1294 2290
12	1	Link failure.	1325
13	1	Session failure.	none
13	2	Reserved.	
13	3	Connection failure for parallel session setup.	none
50	1	Local session limit has been reached.	1496

Table 8-1. $STATUS/D Codes and Error Message Numbers (Continued)

S	SD	Description	EMN
50	2	Local conversation limit has been reached.	1495
50	3	Server process subsystem could not be started because APSY is not initialized.	
50	4	Subsystem is not available. The requested server process subsystem is not defined in CCASYS.	
50	5	Input buffer too small for SUBSYSPARM. The SUBSYSPARM value specified in the process definition was larger than the size in the IODEV 27 command line.	
50	6	Synchronization level not supported by local process. Requesting process and local process must agree on CONFIRM or NOCONFIRM.	
50	7	Server security violation.	
51	1	Server process not available (from client's point of view only).	2031
51	2	Synchronization level not supported by server LU (from client's point of view only). The server process definition does not specify the same synchronization level as that of the local process (CONFIRM/NONCONFIRM).	2031
52	1	Link failure.	none
53	1	Session failure. Check sense code in $ERRMSG.	1432
53	2	Timeout duration exceeded waiting for response from partner.	1968
53	3	Session Bind rejected due to incorrect session parameters.	none
53	4	Unexpected conversation end.	

* Server message

** Client message

A

Horizon Conversation States and Statements

Overview

The communication statements (verbs) that an application program may issue at any given point in the conversation depend upon the current state of the conversation. The Model 204 Horizon conversation states are described briefly below. Table A-1 on page 148 lists the Horizon statements that can be issued in each state.

Conversation states

The following conversation states are used in Horizon:

Reset	The application program may allocate a conversation (open a process).
Send	The application program may send data, request confirmation, or (stop sending and) receive data.
Receive	The application program may receive data from the partner application program or request permission to send.
Confirm	The application program may comply with the request for confirmation. (It may also request permission to send later, after the confirmation.)
Close	The application program may deallocate the conversation (close the process) with the remote application program.

Conversation Statement/State Dependencies

The matrix in Table A-1 lists the Horizon statements that can be issued in a given conversation state.

“No” a state check is incurred because the statement is not valid in that state. State checks are issued only when a process is already open. The dashes in the matrix mean that some error other than a state check is incurred. N/A’s (not applicable) are entered for TEST RECEIPT and WAIT FOR RECEIPT because they are not state dependent.

Table A-1. Statement and state dependencies

Reset	Send	Receive	Confirm	Close
OPEN PROCESS	Yes	-	-	-
CONFIRM	-	Yes	No	No
CONFIRMED	-	No	No	Yes
CLOSE PROCESS	-	Yes	Yes	Yes
FLUSH	-	Yes	No	No
INVITE	-	Yes	No	No
QUERY PROCESS	Yes	Yes	Yes	Yes
RECEIVE	-	Yes	Yes	No
SIGNAL	-	No	Yes	Yes
SEND	-	Yes	No	No
SEND ERROR	-	Yes	Yes	Yes
TEST RECEIPT	N/A	N/A	N/A	N/A
WAIT FOR RECEIPT	N/A	N/A	N/A	N/A

B

LU 6.2 Verb Set Equivalences

Overview

In terms of the LU 6.2 architectural standard, Horizon implements LU 6.2 “mapped conversation” support, using the “base subset” of LU 6.2 options. In addition, some optional verb sets are implemented: explicit flush, PIP data, attribute query, and security.

The tables in the following sections map Horizon programming statements and their parameters and status codes to corresponding terminology used in standard LU 6.2 architectural documents. The SOUL column of the tables is formatted according to the following conventions:

- Items in full capitals are keyword parameters.
- Items in lowercase are positional parameters.
- Items in parentheses are parameters on either DEFINE commands or network control commands; the command name is given in initial capitals.

LU 6.2 and Horizon verb and parameter equivalences

Table B-1 maps LU 6.2 verbs and parameters to SOUL statements and parameters.

Table B-1. LU 6.2 and Horizon Equivalences

LU 6.2 Verbs and Parameters	SOUL Statements and Parameters
MC_ALLOCATE	OPEN PROCESS
LU_NAME(other)	(REMOTEID on Processgroup);
	pointed to optionally by
	AT on OPEN PROCESS
MODE_NAME	(MODENAME on Process)
TPN	(PARTNER on Process)
RETURN_CONTROL	
WHEN_SESSION_ALLOCATED	[default]
SYNC_LEVEL	
NONE	(NOCONFIRM on Process)
CONFIRM	CONFIRM on Process)
SECURITY	
NONE	[default}
SAME	(UIDSOURCE=CURRENT on
	Process,
PGM(USER_ID)	LOGIN=TRUST on Processgroup)
PGM(PASSWORD)	USERID
PGM(PROFILE)	PASSWORD
PIP	PROFILE
(YES(var1,var2,...))	
(NO)	INITIAL DATA
RESOURCE	[default]
RETURN_CODE	processname|CID
	$STATUS and $STATUSD
MC_CONFIRM	CONFIRM
RESOURCE	cid
RETURN_CODE	$STATUS and $STATUSD
REQUEST_TO_SEND_RECEIVED	REQSEND
MC_CONFIRMED	CONFIRMED
RESOURCE	cid
MC_DEALLOCATE	CLOSE PROCESS
RESOURCE	cid
TYPE	
SYNC_LEVEL	SYNCLEV
FLUSH	FLUSH
CONFIRM	CONFIRM
LOCAL	[implicit]
ABEND	ERROR
RETURN_CODE	$STATUS and $STATUSD
MC_FLUSH	FLUSH PROCESS
RESOURCE	cid
MC_POST_ON_RECEIPT	*included in INVITE*

Table B-1. LU 6.2 and Horizon Equivalences (Continued)

LU 6.2 Verbs and Parameters	SOUL Statements and Parameters
MC_PREPARE_TO_RECEIVE	INVITE
RESOURCE	cid
TYPE	
SYNC_LEVEL	SYNCLEVEL
FLUSH	FLUSH
CONFIRM	CONFIRM
LOCKS	
SHORT	[default]
RETURN_CODE	$STATUS and $STATUSD
MC_GET_ATTRIBUTES	QUERY PROCESS
RESOURCE	cid
OWN_FULLY_QUALIFIED_LU_NAME	(LOCAL ID returned by Monitor Link)
PARTNER_LU_NAME	REMOTEID
	REMOTEID
PARTNER_FULLY_QUALIFIED_LU_NAME	MODENAME
	SYNCLEVEL
MODE_NAME	
SYNC_LEVEL	
MC_RECEIVE_AND_WAIT	RECEIVE
RESOURCE	FROM cid
DATA	%variable|IMAGE imagename
LENGTH	[implicit]
RETURN_CODE	$STATUS and $STATUSD
WHAT_RECEIVED	RESULT
MC_REQUEST_TO_SEND	SIGNAL
RESOURCE	cid
MC_SEND_DATA	SEND
RESOURCE	TO cid
DATA	%variable|’string’|IMAGE imagename
LENGTH	[implicit]
RETURN_CODE	$STATUS and $STATUSD
REQUEST_TO_SEND_RECEIVED	REQSEND
MC_SEND_ERROR	SEND ERROR
RESOURCE	TO cid
RETURN_CODE	$STATUS and $STATUSD
REQUEST_TO_SEND_RECEIVED	REQSEND
MC_TEST	TEST RECEIPT
RESOURCE	cid
TEST	
POSTED	[default]
RETURN_CODE	$STATUS and $STATUSD

Table B-1. LU 6.2 and Horizon Equivalences (Continued)

LU 6.2 Verbs and Parameters	SOUL Statements and Parameters
WAIT	WAIT FOR RECEIPT
	receipt cid
RESOURCE_LIST	*or*
	ANY receipt
RESOURCE_POSTED	RETURN
RETURN_CODE	$STATUS and $STATUSD

LU 6.2 verb equivalences

Table B-2 maps LU 6.2 verbs to Horizon SOUL statements, CICS commands, and AT&T LU 6.2 Facility commands.

Table B-2. LU 6.2 Verb Equivalences

LU 6.2 Verbs	Horizon SOUL Statements	CICS Commands
MC_ALLOCATE	OPEN PROCESS	EXEC CICS ALLOCATE EXEC CICS CONNECT PROCESS
MC_CONFIRM	CONFIRM	
MC_CONFIRMED	CONFIRMED	
MC_DEALLOCATE	CLOSE PROCESS	EXEC CICS RETURN *or* EXEC CICS SEND LAST *or* EXEC CICS ISSUE ABEND
MC_FLUSH	FLUSH PROCESS	EXEC CICS WAIT
MC_GET_ATTRIBUTES	QUERY PROCESS	EXEC CICS EXTRACT PROCESS
MC_PREPARE_TO_RECEIVE	INVITE	EXEC CICS ISSUE SIGNAL
MC_POST_ON_RECEIPT		
MC_RECEIVE_AND_WAIT	RECEIVE	EXEC CICS RECEIVE
MC_REQUEST_TO_SEND	SIGNAL	
MC_SEND_DATA	SEND	EXEC CICS SEND
MC_SEND_ERROR	SEND ERROR	EXEC CICS ISSUE ERROR
MC_TEST	TEST RECEIPT	
WAIT	WAIT FOR RECEIPT	
OS/2 APPC	**Horizon SOUL Statements**	**AT&T LU 6.2 Facility**
MC_ALLOCATE	OPEN PROCESS	malcnv
MC_CONFIRM	CONFIRM	mcnfrm
MC_CONFIRMED	CONFIRMED	mcnfrmed
MC_DEALLOCATE	CLOSE PROCESS	mdalcnv
MC_FLUSH	FLUSH PROCESS	mflush
MC_GET_ATTRIBUTES	QUERY PROCESS	mgetadr
MC_GET_TYPE		
MC_PREPARE_TO_RECEIVE	INVITE	mprprev
MC_POST_ON_RECEIPT		mpstrct
MC_RECEIVE_AND_WAIT	RECEIVE	mrcvwt
MC_RECEIVE_IMMEDIATEM		mrcvim
MC_REQUEST_TO_SEND	SIGNAL	mrqssnd
MC_SEND_DATA	SEND	msnddta
MC_SEND_ERROR	SEND ERROR	msnderr
MC_TEST_RTS	TEST RECEIPT	mtestev
	WAIT FOR RECEIPT	

C

SNA Communications Server Performance Tuning for Horizon

Overview

SNA Communications Server (formerly VTAM) network performance tuning includes the monitoring of response times and network loads, and the adjustment of SNA Communications Server, NCP, and application variables. The goal of the tuning is to improve response time and allow for greater capacity.

Although SNA Communications Server tuning necessarily involves the SNA Communications Server system programmer, some of the variables that can be tuned to increase performance have to do with the SNA Communications Server definitions discussed in “Defining the Network to SNA Communications Server” on page 36. Three of those variables (associated with the APPL statement and the mode table) are briefly discussed in this appendix:

- Chaining and RU sizes
- Session-level pacing
- Transmission priority and route selection

Chaining and RU Sizes

In the mode table entry for Model 204/Horizon supplied in “Defining the Network to SNA Communications Server” on page 36, the RUSIZES parameter (maximum secondary LU send RU length, and maximum primary LU send RU length) is coded to restrict RU lengths in both directions on the session to 2K (2048) bytes. The two 1-byte values are coded in normalized floating-point notation, in the form:

```
m * 2n
```

where *m* is the first digit of the value and *n* is the second digit. Thus, x’88’ is 8 * 28, or 2048. These values are suitable for most applications.

The restrictions on RU sizes for a session compel the session partners to segment messages exceeding the maximum into smaller elements “chained” together.

When to chain

Chaining is a common means of achieving better performance in SNA networks for the particular circumstances described below. When these circumstances apply, it may be advisable to lower the RU maximums in order to attain more chaining.

The first of these circumstances pertains to network topology and the second pertains to Model 204 system resources.

- When a network contains remote NCPs (that is, multiple “hops” between source and destination), connected by relatively low-speed lines, transmission delays can be greatly reduced by ensuring that the message units being transmitted are small. Chaining of application data accomplishes this since one chain element at a time is transmitted.

- When the Model 204 ONLINE running Horizon conversations is virtual-storage constrained and at the same time required to support a significant number of conversations concurrently, efforts should be taken to reduce the buffer requirement for those conversations. For each active, concurrent conversation, Horizon allocates as a common buffer (for data sent and received) the larger of the following values:

  – Maximum primary send RU size

  – DEFINE PROCESS DATALEN= parameter value

  This buffer is allocated for the duration of the conversation.

  For example, if the largest message expected from a partner across all processes is 1K bytes and the table entry still calls for a maximum send RU size of 2K, the buffer allocated for each conversation in Model 204 is at least 1K larger than actually needed. If it becomes necessary to conserve virtual storage, setting maximum RU sizes to 1K in the mode table entry for Model 204 causes Model 204 to allocate just 1K for each buffer, resulting in a savings of 1K per active conversation.

Note: Unless you are sure that the expected messages in the two directions between the ONLINEs will be unequal in length, keep the values for both secondary and primary LU the same, rather than paring down the values separately as much as possible: Horizon returns a “DATA TRUNCATED” error code to the application process when the process receives an application message larger than the buffer it has allocated. If, on the other hand, the sending side has a maximum RU equal to this receiving side's buffer, the large message is sent as a chain of smaller elements, each of which fits into the receiver's buffer.

Chaining's cost

Message chaining has a cost: CPU usage. Chaining means multiple sends per message, whereas only a single send is required when the message is sent in its entirety as a single unit. More CPU cycles are needed to accomplish the multiple sends. The network tuner must check to make sure that CPU usage by Model 204 has not increased significantly by the level of chaining introduced.

Session-Level Pacing

Session-level pacing is an important means of protecting internal network components from overload by individual users. An operand of the APPL statement and an operand of the mode entry statement combine to specify the session-level pacing parameters.

Pacing parameter settings

In the samples provided in "Defining the Network to SNA Communications Server" on page 36, the flow control, or pacing, variables are set to 5 in both directions. The SNA Communications Server on the sending side forwards no more than five messages (message chain elements, actually) on a single session involving this Horizon "link" to the SNA Communications Server of the receiving side, until an internal acknowledgment (a "pacing response") is received from the sending SNA Communications Server.

Usually there is no need to alter this value. The setting of 5, in conjunction with the large maximum RU size of 2K, protects the network from long chains of data without slowing down the flow of average-length data.

Modification of pacing parameter values

Pacing values may be reconsidered if, perhaps for reasons outlined in the discussion above on chaining, the maximum RU sizes are reset to values much smaller than the average application message length. The pacing values should then be set higher to avoid unnecessary throttling of the conversations.

Transmission Priority and Route Selection

This network performance topic is directed to the network planner, the person doing the initial preparation for inter-computer traffic. The objective is to prevent network batch transmissions from disturbing shorter, network interactive transmissions.

Horizon transmissions are usually the interactive type: a user triggers a remote process by opening a local process through terminal-entered commands and waits at the terminal for the remote process to complete. If interactive transmissions experience transient delays, at least some users are disturbed. Transient delays of batch transmissions (such as file transfer), however, are unobserved and overall turnaround time is instead the performance criterion.

Transmission priority refers to NCP’s ability to prioritize transmission across NCP-to-NCP lines. Route selection refers to the ability of both SNA Communications Server and NCP to force different sorts of traffic onto logically distinct and independently managed “virtual routes.” Network planners can take advantage of these facilities to allow interactive traffic to be transmitted ahead of, or on lines different from, batch traffic.

Assigning transmission priority and route selection

When a computer network sporadically transmits large amounts of data of a systems nature (such as job output) between the computers, an operating-system level subsystem, such as JES or RSCS, is typically used. The network planner uses the mode table entries for those subsystems to associate the batch traffic they send with yet another network table, the class of service (COS) table. The COS= operand on the MODEENT macro refers to the name of the COS table entry. Entries in the COS table determine transmission priority and route preference.

A special entry can be assigned to these batch transmission subsystems so that their traffic flows at a lower priority than, or separately from, Horizon interactive traffic. The Model 204/Horizon mode table entry (and entries for online systems generally) can then assume, by omitting the COS= operand, an association with the “default COS table entry,” which is coded so as to designate a high priority transmission.

Horizon batch transmissions

Horizon processes can also involve batch transmissions. Applications can be written, for example, to periodically transfer Model 204 audit trails between systems for merger at a central point.

To separate such applications from interactive ones, the performance tuning technique is to include a COS= operand in a new mode table entry for the Horizon systems involved in the application. The Model 204 DEFINE PROCESSGROUP MODENAME= parameter can then be used to call for a session with the special routing characteristics desired when the audit-trail transfer application process is started.

D

Connecting to Non-Model 204 Systems

Connecting to CICS

Compared to a Model 204-to-Model 204 Horizon conversation, a Model 204-to-CICS Horizon conversation requires some additional conceptual understanding (of the CICS side) but little additional programming effort for the Model 204 side. The Model 204 network administration adjustments to CICS protocols are largely transparent to the programmer. There are no changes to the Horizon conversation interface "verbs."

This Overview presents a discussion of how Horizon and CICS connect. The rest of this section describes how to prepare the SNA Communications Server (formerly VTAM), Model 204, and CICS network definitions and how to implement and maintain the operation of the connection.

Horizon connectivity versus CICS Interface connectivity

Horizon connectivity to CICS differs in several respects from the CRAM-based Model 204 CICS (terminal) Interface:

- The Model 204 CICS Interface operates only when the Model 204 and CICS systems are resident on the same z/OS machine. Horizon connects to CICS on the same or on a widely separated machine.
- Special Model 204 software or SVCs or both must be installed in CICS and z/OS for the CICS Interface, but not for Horizon. In

Horizon, software in Model 204 communicates peer-to-peer with native CICS software.

- The CICS Interface is primarily a means for a CICS-controlled terminal user to run programs in Model 204. Horizon is a means for CICS and Model 204 application programs to converse and share data as peers.
- The application programming in CICS that corresponds to Horizon SOUL programming is done with a set of extensions to CICS Command Level programming. Like Horizon SOUL statements, these CICS commands each correspond, often very closely, to an APPC verb.
- Horizon connectivity is through SNA Communications Server. Both Model 204 and CICS therefore must implement “logical units” to serve as their entry points into the SNA network. In Model 204, this entry point is the Horizon “link,” of which there may be several. In CICS, this entry point is the single LU that also serves the CICS terminal handler.

Figure D-1 shows the components of the Horizon/CICS environment. LU 6.2 conversations between CICS transactions and Model 204 procedures can be started from a terminal controlled by either system.

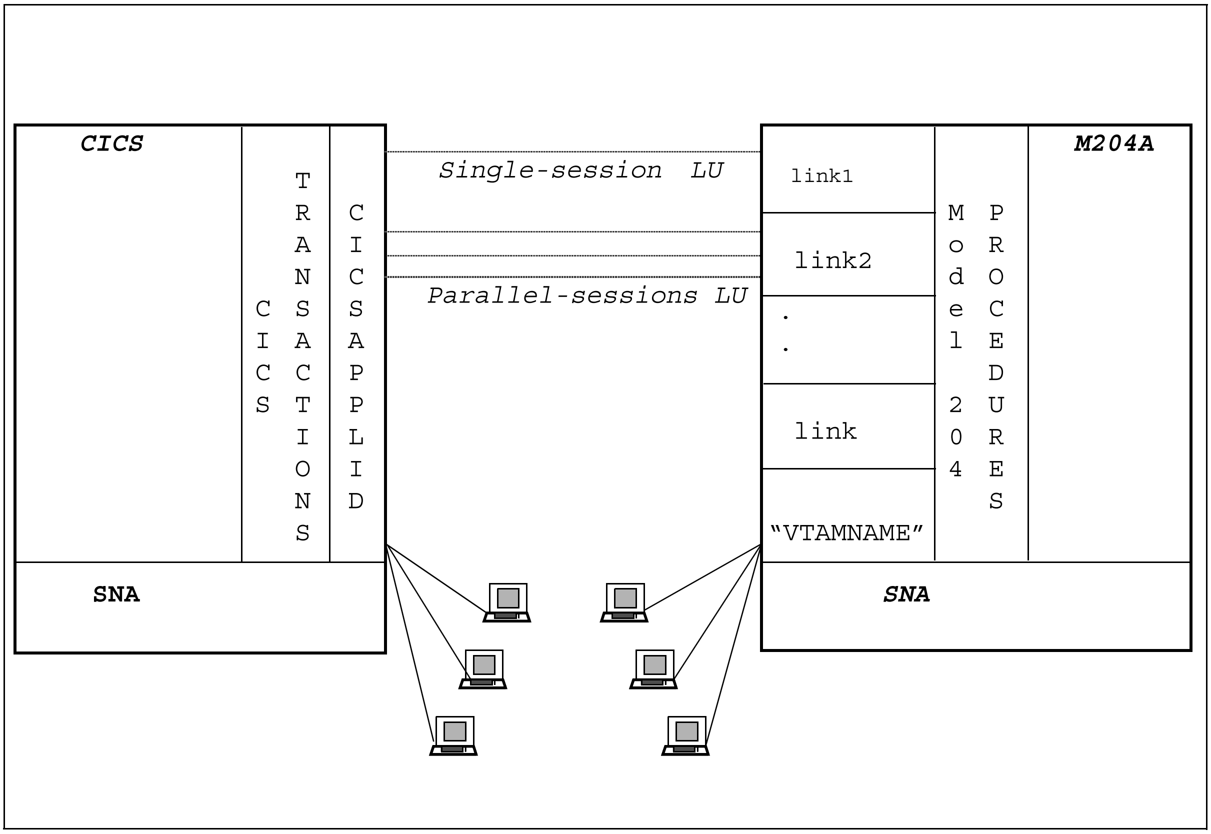


Figure D-1 The Horizon/CICS environment

One LU for CICS; multiple LUs for Model 204

In Figure D-1, CICS has one LU and Model 204 has several. This does not imply any essential difference between the two systems. Model 204's ability to provide more than one link simply offers some extra flexibility in system management. The cost is some added complexity in system administration.

For example, you can assign one link for applications transferring large amounts of data between the two systems per request, and you can assign one link for applications with heavy request volume but relatively short data transfers per request. Such a scheme optimizes response time. The cost is that you have to maintain extra definitions for the second link and its associated processgroups.

Figure D-1 shows many sessions in progress simultaneously between CICS and one of the Model 204 links. Such a configuration is called "parallel sessions." A simpler configuration is possible: each Model 204 link can support just one session, with users employing that session for program-to-program conversations in series, that is, one after the other.

Note: Separate additional links per CICS region are required if Model 204 connects to more than one CICS region simultaneously. For more information about this, see "Network definition requirements" on page 164.

Supporting CICS parallel sessions

To support the parallel sessions configuration between CICS and Model 204, you must prepare "CNOS" support: support for special intersystem protocols used by the system software for setting up, terminating, and dynamically controlling the sessions between the two systems.

CNOS support preparation is described in Appendix E. It includes updating the SNA Communications Server "logmode" definitions and APPL statements, as well as updating definition commands in Model 204 and CICS.

SNA Communications Server Definition Statements

This section describes the SNA Communications Server network preparation required for LU 6.2 support: updating the SNA Communications Server "logmode" definitions, determining session parameters, and coding the APPL statements.

Preparing the log mode definitions

Mode table configuration, including an example of recommended mode table entry values for Horizon, is discussed in "Defining the Network to SNA Communications Server" on page 36.

The following are recommendations for mode table configuration:

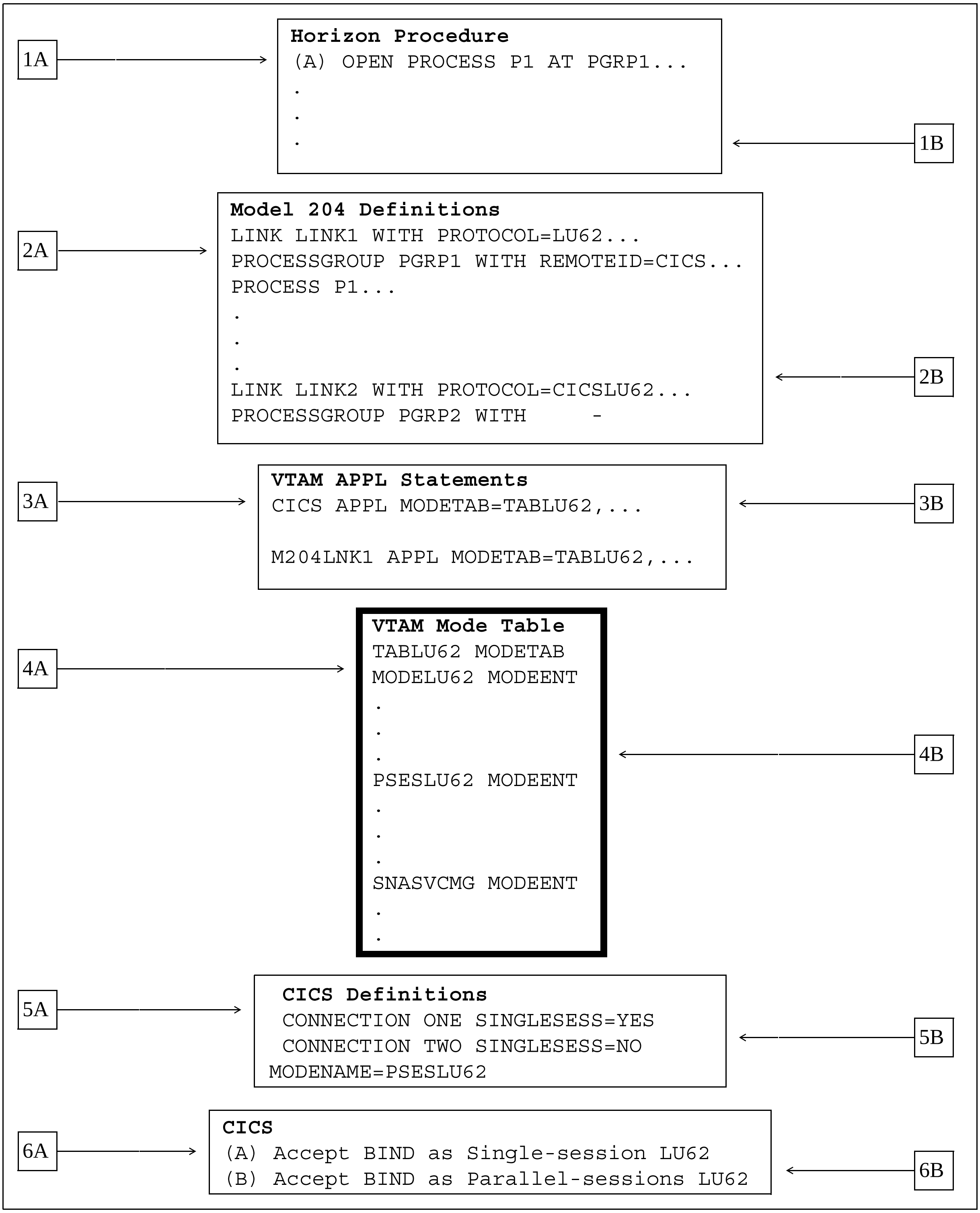
- Use *only* one table for all LU 6.2 sessions, between all systems. Let this table’s first entry—which serves as the default entry—be the most widely applicable one, such as the one with the values for single-session support recommended in Chapter 3.
- If using CNOS sessions, include an SNASVCMG entry and an entry with parallel session support somewhere in the table.
- Code the table’s name on the MODETAB parameter of the APPL statements for both CICS and for each of the Horizon links in MODEL 204.
- Omit the DLOGMOD parameter.
- For all but CNOS connections, omit the MODENAME parameter on both the Horizon PROCESSGROUP and the CICS CONNECTION definitions.
- For CNOS connections, include MODENAME in the Horizon SESSIONGROUP and the CICS CONNECTION definitions. This MODENAME will be the name of the CNOS parallel-sessions entry in the single, common mode table for LU 6.2.
- Code the table’s name on the MODETAB parameter of the APPL statements for both CICS and for each of the Horizon links in Model 204.
- Omit the DLOGMOD parameter.
- For all but CNOS connections, omit the MODENAME parameter on both the Horizon PROCESSGROUP and the CICS CONNECTION definitions.
- For CNOS connections, include MODENAME in the Horizon PROCESSGROUP and the CICS CONNECTION definitions. This MODENAME will be the name of the parallel-sessions entry in the single, common mode table for LU 6.2.

Determining session characteristics

When sessions are started from Model 204, the SNA Communications Server session parameters are determined by reference to the Model 204 network entity definitions, the APPL statements for CICS, and the common mode table for Horizon and CICS LU 6.2. Finally, the session parameters are checked by CICS against CICS definitions before the bind is accepted.

Figure D-2 on page 163 shows the steps in the bind selection process when Model 204 initiates the process. The mode table configuration in Figure D-2 allows a common, default entry to be chosen when no MODENAME is specified in the Model 204 definitions (sequence (A) in figure) and a specific entry for CICS parallel-session support (sequence (B) in figure) when the entry is specified on the DEFINE PROCESSGROUP command. Such a configuration is recommended for connections between Horizon and CICS.

When sessions are started from CICS, the bind process steps in Figure D-2 are reversed.



Horizon Procedure

```
(A) OPEN PROCESS P1 AT PGRP1...
.
.
.
```

Model 204 Definitions

```
LINK LINK1 WITH PROTOCOL=LU62...
PROCESSGROUP PGRP1 WITH REMOTEID=CICS...
PROCESS P1...
.
.
.
LINK LINK2 WITH PROTOCOL=CICSLU62...
PROCESSGROUP PGRP2 WITH     -
```

VTAM APPL Statements

```
CICS APPL MODETAB=TABLU62,...

M204LNK1 APPL MODETAB=TABLU62,...
```

VTAM Mode Table

```
TABLU62 MODETAB
MODELU62 MODEENT
.
.
.
PSESLU62 MODEENT
.
.
.
SNASVCMG MODEENT
.
.
```

CICS Definitions

```
 CONNECTION ONE SINGLESESS=YES
 CONNECTION TWO SINGLESESS=NO
MODENAME=PSESLU62
```

CICS

```
(A) Accept BIND as Single-session LU62
(B) Accept BIND as Parallel-sessions LU62
```

Figure D-2 SNA Communications Server Session Parameter Selection

Following is a brief description of the sequence in Figure D-2 on page 163:

Step	Description
(1)	The session is started from Model 204. The processgroup name specified after the AT parameter indicates where (with which processgroup and link definition) to find the session characteristics.
(2)	MODENAME, if any, points to a specific mode table entry. If MODENAME is not specified (as shown), the default mode table entry is used. REMOTEID indicates the CICS LU.
(3)	The SNA Communications Server APPL statement for the CICS LU points to the common mode table, TABLU62.
(4)	TABLU62, the common mode table, the name of which precedes the list of entries. The first entry in the list, here MODELU62, is the default. As recommended in “Preparing the log mode definitions” on page 161, the default entry specified is the one with the session parameters for single-session support.
(5)	The CICS connection definitions are checked by CICS when the Bind arrives.
(6)	The result is a Bind for single-session support.

Coding the APPL statement

The APPL statements for the Horizon links to CICS are identical to those for Model 204-Model 204 links.

Typically, you have to modify the CICS APPL statement to include parameters relevant to LU 6.2 support (PARSESS=YES, etc.). You need to add a MODETAB parameter, since other than LU 6.2 connections, CICS will not have been the secondary LU in any application. The MODETAB parameter should point to the common LU 6.2 table, as stated in the mode table configuration recommendations in “Preparing the log mode definitions” on page 161.

Model 204-Side Definition Commands

The DEFINE command specifications prepare the Model 204-side of the connection to CICS. The CICS and Model 204 staffs must coordinate at definition time with regard to the names used (for REMOTEID, PARTNER, and so on).

For information about the DEFINE commands for CNOS parallel sessions, see Appendix E.

Network definition requirements

The DEFINE command specifications listed below are modified or require special attention for CICS single-session support.

- DEFINE LINK command PROTOCOL parameter

  One Horizon link must be reserved for each CICS system to which the Model 204 system connects. Each of these links appears to CICS to be a separate system.

- DEFINE PROCESSGROUP command MODENAME parameter

  The MODENAME parameter is required on the definition for the processgroup whose REMOTEID parameter points to the CICS partner. More than one processgroup can be defined for this link, but the MODENAME (and REMOTEID) must be the same on each.

  (On the CICS side, CICS online resource definition or CICS DFHTCT macros specify the LU/mode definition (CICS “connection” and “sessions” definitions) which identifies the Model 204 system. If more than one processgroup, as mentioned above, is defined on the Model 204 side for the Horizon link, the CICS definitions do not change in any way.)

- DEFINE PROCESS command PARTNER parameter

  The PARTNER parameter value must be the four-character CICS transaction ID in which the CICS partner program is to run.

- DEFINE PROCESS command PARTNER parameter

  The PARTNER parameter value must be the four-character CICS transaction ID in which the CICS partner program is to run.

- DEFINE LINK command SESSIONS parameter and DEFINE PROCESSGROUP command INLIMIT and OUTLIMIT parameters

  The CICS and Model 204 staffs must coordinate with regard to initial values for session limits specified on the Model 204 DEFINE PROCESSGROUP command and on the CICS DEFINE SESSIONS. It is suggested that the total be set high and that it be divided equally between inbound and outbound sessions.

  That is, let SESSIONS on the DEFINE LINK command be high, and let the INLIMIT value equal the OUTLIMIT value on the DEFINE PROCESSGROUP command. These Model 204-side values are to correspond to two values on the CICS SESSIONS MAXIMUM definition, as follows:

- The total number of application sessions allowed with the named Model 204 link is indicated by the first MAXIMUM value in CICS and by the sum of INLIMIT and OUTLIMIT in Model 204.

- The number within the above total reserved for sessions on which CICS is assured to win any contention for a chance to start an application conversation is indicated by the second MAXIMUM value in CICS and by INLIMIT in Model 204.

The difference between the two CICS MAXIMUM values is the number of sessions on which CICS defers to Model 204 when both wish to use the session.

Note: The Model 204 DEFINE LINK SESSIONS value should include two extra control sessions which are not used for applications. None of the values on the processgroup definition (and none of the CICS values) includes these two extra sessions.

CICS-Side Definition Commands

The CICS *Inter-communication Facilities Guide* explains the statements and parameters needed to configure LU 6.2 connections in CICS. Especially relevant to CICS connections to Model 204 are the CICS CONNECTION and SESSIONS definitions. These definitions, taken together, roughly correspond to the Model 204 PROCESSGROUP definition.

LU 6.2 network definition considerations

This section discusses aspects of the CONNECTION and SESSIONS definitions that are important for Model 204 LU 6.2 connections.

Parameter names used are for CICS RDO.

- MODENAME (on SESSIONS) is required for both single- and parallel-session connections.
- SINGLESESS (on CONNECTION) determines whether or not parallel sessions are supported (SINGLESESS=N invokes the support).
- BINDPASSWORD (on CONNECTION) must be left blank.
- There are two values required on MAXIMUM (on SESSIONS). The first of these refers to the total number of concurrent application sessions allowed with any one Model 204 “link.” The second specifies the number of sessions out of the total on which CICS will be assured the ability to start a conversation.
- RECEIVESIZE and SENDSIZE (on CONNECTIONS) are the maximum values CICS allows for RUSIZES on a Bind from another system. CICS negotiates the Bind values downward to these values. For ease of administration, code all values the same: the two values for RUSIZES in the logmode entry and these two CICS parameters. 2048 is a good value.

Security considerations

The levels of security support for conversations invoked from the Model 204 side are specified on the ATTACHSEC parameter (on CONNECTION). They should correspond as follows in Table D-1 to Horizon security values:

Table D-1. CICS LU 6.2 Security Support Compared to Horizon

CICS side	Model 204 side	
`LOCAL`	`UIDSOURCE=NONE`	(no security)

`IDENTIFY`	`LOGIN=TRUST +`
	`UIDSOURCE=OPEN or`
	`UIDSOURCE=CURRENT`
		(Model 204 sends user ID but no password)

`VERIFY`	`UIDSOURCE=OPEN or`
	`UIDSOURCE=CURRENT +`
	`OPEN PROCESS PASSWORD`
		(Model 204 sends user ID and password

Note: For conversations invoked on the CICS side, the security level is always equivalent to “identify”: CICS always sends a user ID but **never** sends a password.

Specifying the Model 204 partner program

In CICS, the partner program name for conversations invoked from the CICS side is not specified on a definition statement, but rather in the application program’s EXEC CICS CONNECT command. The PROCNAME parameter on this command must be the appropriate process name in Model 204.

Note: CICS insists that the PROCNAME parameter name be four characters in length. Hence, the name of a server process invoked from CICS also must be four characters in length.

Maintaining Operations

This section describes the principal operating functions for each side in Horizon to CICS conversations: establishing the connection, changing session limits, and executing an orderly shutdown.

Establishing the connection

Once the Horizon link is open and the CICS connection definitions are put “in service,” you can set up the connection between the two systems. “In service” status in CICS does not involve ACB open as does Horizon’s OPEN LINK, but is similar to “started” status for the link in Horizon. Like link opening in Horizon, however, it can be configured to occur automatically at system startup, instead of by doing CEMT I CONN followed by ACQ.

Since you can initiate the connection setup from either system, choose one system as the focal point for control of connection. Using one system as the focal point simplifies the establishment and takedown of the connection.

When Model 204 is the focal point

If Model 204 is designated as the focal point, the first outbound process of the run that is opened on the link, before its open completes, accomplishes the establishment of the connection. The Model 204 system administrator has no special commands to issue.

When this first conversation opens, the MONITOR LINK command display shows two control sessions active on the link in addition to the application session. “X” in the session FLGS (flags) column on the display indicates these control sessions. “S” indicates that Model 204 was the source of the connection. “T” indicates that CICS started the connection and Model 204 was, in CNOS terminology, the connection “target.”

The audit trail shows a message stating the names of the link, processgroup (for the application process just opened), and CICS LU name and that the connection is established. If the SOUL program issues $ERRMSG, the same message is displayed as that on the audit trail. Otherwise, the connection establishment process is transparent to the program.

Should the connection attempt fail at any time during set up of the control sessions or during CNOS Initialize exchange, the status of the connection is reset entirely and a second conversation attempt is allowed to try the connection process again. Failure audit messages record the attempted connection and the reason for the failure. The program attempting the first OPEN PROCESS may detect a failure in the connection process as a $STATUS/$STATUSD code of 13/3 or 53/3.

When CICS is the focal point

If CICS is the focal point, the CICS operator must “bring up the link” to Horizon as a regular, system startup activity. CICS log messages record the session limits specified in the CNOS Initialize exchange. Horizon’s MONITOR LINK will show two idle control sessions active. Either CICS users or Model 204 users may then start conversations.

Changing CNOS session limits

At any time while the connection is up, the CICS operator can decrease the session limits on the connection with a CEMT SET MODENAME command. Horizon will overlay its current session limits with values carried in the CNOS Change command from CICS, and it will return a positive CNOS Reply. The new values are accepted or negotiated according to SNA standards.

In addition, CICS implements the session deactivation aspect of CNOS Change, whereby the CNOS sender designates one partner to be responsible for terminating those sessions in excess of the new limits. Since CICS always designates the CNOS “source” to do the termination, the Horizon system administrator sees some number of sessions lost. The audit messages reporting the lost sessions will not themselves refer to the CNOS Change process. However, they will be preceded by a message stating the receipt of the CNOS Change command from CICS.

For more information about Horizon CNOS support, see Appendix E.

Executing an orderly shutdown

You can bring down a connection, as well as start one up, from either the Model 204 or the CICS side. To close the connection cleanly, however, the CICS side should implement shutdown procedures.

Shutdown from the Model 204 side

From Model 204, CLOSE LINK FORCE or EOJ bring the connection down in an unpredictable manner. Although there are no system abends or the like on the CICS side, and the connection is still re-startable from this point, the CICS log registers some peculiar system level errors.

Shutdown from the CICS side

Orderly shutdown from the CICS side does not involve ACB close. The following sequence of actions occurs:

- The operator “takes down the link” with Model 204 by the CEMT SET CONNECTION RELEASED command.
- Horizon prohibits outbound conversations. A message is audited stating that connection shutdown has begun by request from the CICS side.
- CICS proceeds to drain all active application sessions and then terminates each session. The application sessions are terminated first, the two control sessions last.
- When the control sessions end, the completion of the connection reset process is recorded on the audit trail.

Either side may now restart the connection at any time.

Connecting to VAX/VMS

This section discusses requirements for LU 6.2 conversations between a VAX/VMS system and Model 204. It is assumed that members of both staffs will coordinate with regard to the names and values used to meet the network definition and data transmission protocols. Information here is provided according to responsibility: there are sections for SNA Communications Server system programmers, system managers, and application programmers, or their equivalents.

The *DECnet/SNA Guide to IBM Parameters* is a valuable reference source for the information in this section.

Digital products needed to support the connection

On each VAX or MicroVAX involved in a program-to-program application with Model 204, install Digital's LU 6.2 software product, "DECnet/SNA VMS APPC/LU6.2."

Either of two additional Digital products is needed to provide the basic SNA protocol stack, on top of which the LU 6.2 product runs:

- DECnet/SNA Gateway, a software product, is for use when a DECnet network connects the several Digital computers needing LU 6.2 support.
- VMS/SNA, a communications processor box, is for use when these machines are not themselves connected together.

A single Gateway box provides the SNA service to all VAXs on the network, while in standalone mode, each VAX must run the SNA software.

Note: The Gateway function may alternatively be provided by products currently available from other vendors.

SNA Communications Server system programmer considerations

Special SNA Communications Server/NCP definition items are required:

You must choose one of the following methods of customizing the SNA Communications Server/NCP definition for the VAX. If neither of these is entered, the Bind sent by Horizon is rejected; no conversation can take place:

- When defining the VAX's and their LUs to SNA Communications Server/NCP, the VAX requires outbound pacing on the data flow to it from the NCP. Make the alteration to the logmode table for LU 6.2 that is suggested on page 41 of this Horizon guide. Add the following parameter setting:

  ```
  SRCVPAC=4.
  ```

- Alternatively, the logmode table entry can remain as suggested (with SRCVPAC defaulting to 0) if the following pacing value is entered on the LU macro(s) for the VAX in the NCP:

  ```
  PACING=4
  ```

Use the following definition statements to define the Digital interconnect system to ACF/NCP:

- GROUP
- If the DIAL operand is YES, the group line must be identified as half-duplex in the DECnet/SNA configuration file for SNA lines.
- LINE
- If the GROUP definition DIAL operand is YES:
- Set ADDRESS to (lnbr,HALF), where lnbr is the relative line number of the communication line.
- Set NRZI to NO for the DECnet/SNA Gateway.
- The NRZI setting depends on the communication device. Consult VMS/SNA documentation for the setting to use for your device.
- PU
- If the GROUP definition DIAL operand is NO:
- The ADDR value must be set in the DECnet/SNA configuration file for SNA lines.
- MAXDATA must be equal to SEGMENT SIZE in both DECnet/SNA and VMS/SNA.
- DEC strongly recommends that MAXOUT be set to 7.
- LU
- In addition to the logmode table modifications for LU 6.2 suggested on page 41 of this Horizon guide:
- For LOCADDR, DECnet/SNA Gateway supports a range of values of 1 to 128; VMS/SNA supports a range of values of 1 to 64.

Some items in the IBM or Horizon definitions pertaining to the SNA connection must be repeated in the Digital definitions. In the VMS/SNA software product, such definitions are done with the “SNA Network Management” facility.

To use this facility, enter “run sys$system:snanm”. At the SNANNM prompt, enter SET ACCESS NAME. Prompts follow for the input of five items, three of which you enter or modify:

- APPLICATION refers to the SNA Communications Server APPL name used by the Horizon link, and it must correspond to the Horizon LOCALID on the DEFINE LINK statement.
- LOGON refers to the logmode table entry to be used on the session. The name entered here must correspond to the name on the MODEENT statement for the logmode table entry in SNA Communications Server that has been set up for a VAX-to-Horizon session.

- ACCESS NAME is the tag for this set of definition statements. You refer to this name in the VAX program.

System manager considerations

This section uses a sample program from the *DECnet/SNA VMS APPC/LU6.2 Programming Interface* as a reference point for the discussion of network definition requirements.

Implementing the sample application program

Digital program-to-program communications applications can be coded in Fortran, Pascal, Basic, Cobol, PL/I, C, or VAX Macro. A sample is provided in each of these languages in the Digital manual for LU6.2. The following notes may help to implement the sample.

The parameters on the DEFINE_REMOTE verb in the sample require care:

- ACC_NAME is the name of the definition set entered through the SNA management facility (see above). When this parameter is included, none of the parameters listed as options after it (not even (PLUNAME) need be included. The name of the Horizon LU is obtained instead from the “access” definitions.

  There are, however, two required parameters containing the term LUNAME. These are actually two aliases that the VAX programmer can use for the Horizon LOCALID and they may or may not be identical to the Horizon LOCALID. To avoid confusion, code the name of the Horizon LOCALID for both of these (as well as for APPLICATION in the SNANM facility.)
- The parameter NODE-NAME on the DEFINE_REMOTE verb refers to the DECnet designation for the DECnet/SNA Gateway, if there is one. If VMS/SNA is used (that is, if a VAX is connecting stand-alone to Model 204), the value for NODE_NAME must be 0.
- The parameter TPN_NAME on the DEFINE_REMOTE verb refers to the Horizon server process name. This is the only place where the Horizon process name can (and must) be indicated.

Application programmer considerations

This section uses the sample programs from the *DECnet/SNA VMS APPC/LU6.2 Programming Interface* as a reference point for the discussion of conversation programming requirements.

Changing the SYNCLEVEL value

The VAX samples allocate the conversation with SYNCLEVEL of CONFIRM. Horizon, however, uses SYNCLEVEL as the default for CONFIRM/NOCONFIRM on its CLOSE PROCESS. With SYNCLEVEL of

CONFIRM, the VAX side is requested to confirm receipt of the close request from Horizon before it can close itself.

But the samples have no confirmation routine. You must change the constant for SYNCLEVEL (at the top of the samples) to “NONE” to run a program without confirmation exchanges.

If you keep the SYNCLEVEL of CONFIRM, you have to add a CONFIRMED verb to the VAX program. Issue this CONFIRMED verb when RECEIVE returns a CONFIRM_DEALLOCATE status (which happens when the Horizon program issues CLOSE PROCESS).

Connecting to OS/2 EE

This section discusses requirements for LU 6.2 conversations between OS/2 EE and Model 204. It is assumed that members of both staffs will coordinate with regard to the names and values used to meet the network definition and data transmission protocols. Information here is provided according to responsibility: there are sections for SNA Communications Server system programmers and application programmers, or their equivalents.

OS/2 products needed to support the connection

On each workstation involved in a program-to-program application with Model 204, OS/2 EE Communications Manager must be installed. This includes the APPC files required to compile programs that issue APPC requests.

A variety of physical link types are supported for connecting to the mainframe, among them SDLC and Token Ring (to channel-attached TR gateway). In addition, APPC is supported by OS/2 in an SNA Gateway configuration, where only one workstation need provide the physical link to the mainframe.

SNA Communications Server system programmer considerations

Special SNA Communications Server/NCP definition items are required:

- When defining the PC’s LUs to SNA Communications Server/NCP, the SNA Communications Server logo screen should not be displayed to the PS/2. This logo (“message 10") should be suppressed. Neglecting to suppress this message causes session binding to fail.

  This logo suppression can be done in either of two ways:

  – Not specifying a USS table in the SNA Communications Server APPLID

  – Pointing to a table that does not include the terminal operator messages

Note: The IBM default USS table (ISTINCDT) does not include an MSG 10.

- On an SDLC line, the SDLC window size (which in NCP is represented by the MAXOUT parameter on the PU macro for the PC LU) must also be entered in Communications Manager under the SDLC DLC ADAPTER PROFILE panel. The two values to adjust are:
  - Send window count
  - Receive window count
- Large RUSIZES should be specified using the RUSIZES in the LOGMODE entry for the APPLID. A value of X'8888', representing 2K, is recommended. Also, in Communications Manager under the SDLC ADAPTER PROFILE panel, the maximum RUSIZE should be adjusted to reflect the 2K value.
- On a switched SDLC line, the NODE ID on the SNA BASE PROFILE in Communications Manager must be in sync with the IDNUM parameter on the PU macro for the IBM PS/2 LU. Also, you must enter a value of "05D" in the IDBLK parameter in the PU macro.
- You can use the recommended logmode table entry shown on page 41 in this Horizon guide.

Application programmer considerations

IBM provides several sample programs written in C, FORTRAN and PL1. For testing, the C file requestor program was customized for Model 204. In this customizing process and APPC coding in general, there are several things to keep in mind:

- On the TP_STARTED APPC command:
  - The lu_alias must be equal to the LU Alias specified on the LOCAL APPC LOGICAL UNIT PROFILE window in Communications Manager.
  - The tp_name should be equal to the name of the program on the workstation (that is, the XXXXXXXX.EXE module).
- As a result of the TP_STARTED command, a tp_id is returned. This tp_id must be used on all future APPC calls.
- On the MC_ALLOCATE APPC command:
  - The plu_alias is equal to the LU alias found on the PARTNER LU PROFILE window in Communications Manager.
  - The tp_name must be equal to the name of the process in Model 204 that is being used as the server routine (this must be presented in EBCDIC).
  - The mode_name must be equal to the Mode Name found on the TRANSMISSION SERVICE MODE PROFILE found in Communications Manager (this must be presented in EBCDIC).
  - Optionally, the pwd and user_id used to logon to Model 204 are included in the MC_ALLOCATE command (these must be presented in

EBCDIC).

- On the MC_SEND_DATA command, any data passed to Model 204 must be converted to EBCDIC before the command is executed.
- On the MC_RECEIVE_AND_WAIT command, any returned data from Model 204 must be converted to ASCII after the command is executed.

Note: The same memory location used for APPC calls is also used for the APPC SV_CONVERT routine. As a result, primary return codes, secondary return codes, and what-received indicators must be either saved or analyzed prior to executing the conversion routine. Failure to do so results in invalid processing due to the examination of returned values that have changed.

- On the SV_CONVERT command, use the SV_G character set. It converts more characters than the SV_AE character set does, and it is user modifiable. The table is under the name supplied on the WORKSTATION PROFILE window in Communications Manager under the Translation Table File Name prompt.

Connecting to UNIX System V and AT&T LU 6.2 Facility

The AT&T LU 6.2 Facility is a software package for the AT&T workstations (3B1s, 3B2s) that run UNIX System V. UNIX System V uses SNA LU 6.2 to communicate with other LU 6.2 supporting platforms.

This section discusses requirements for LU 6.2 conversations between the AT&T LU 6.2 Facility and Model 204. It is assumed that members of both staffs will coordinate with regard to the names and values used to meet the network definition and data transmission protocols. Information here is provided according to responsibility: there are sections for SNA Communications Server system programmers, system managers, and application programmers, or their equivalents.

Products needed to support the connection

- AT&T LU 6.2 Facility
- AT&T 3B1,B2 workstations running UNIX System V.

SNA Communications Server system or network considerations

Special SNA Communications Server/NCP definition items are required:

- The EXCHID on the UNIX system for LOCAL NODE (switched lines only) must match the SNA Communications Server/NCP Major Node definition for PU: that is, EXCHID must equal IDBLK plus IDNUM. For example:

  ```
  If IDBLK=03E and IDNUM=0002, EXCHID=03E0002
  ```

- AT&T LU 6.2 LINE definition parameter MAX_BTU must equal SNA Communications Server/NCP PU’s MAXDATA. If HDX=YES on the AT&T

LINE definition, set DUPLEX to HALF on the SNA Communications Server/NCP LINE definition. If HDX=NO, set DUPLEX to FULL.

- Let NRZI=NO for both the AT&T LU 6.2 LINE definition and the SNA Communications Server/NCP GROUP definition.
- AT&T LU 6.2 entity STATION parameter ADDRESS must equal SNA Communications Server/NCP PU's ADDR.
- The SNA Communications Server/NCP LU pacing parameters VPACING and PACING must match the AT&T MODE definition parameters SPCT and RPCT, respectively.

System manager considerations

The *REMOTEID* parameter on the Model 204 DEFINE PROCESSGROUP command must match the UNIX LOCAL LU *NNAM* parameter. The LOCAL LU is similar to the SNA Communications Server/NCP LU, and both addresses must match.

The UNIX REMOTE LU *NNAM* parameter must match the Model 204 DEFINE LINK command *LOCALID* parameter.

Application programmer considerations

Sample programs used to demonstrate the UNIX System V and AT&T LU 6.2 Facility connection to Model 204 duplicated the C program flow from the *AT&T 3B2 Computer AT&T LU 6.2 Facility Administrator's and Programmer's Guide*, Release 2.0.

ASCII to EBCDIC conversion must be done on the UNIX side.
C subroutines are provided to do this.

If you pass a user ID and password from the UNIX System V workstation to Model 204, avoid mixed case. It is necessary to force UPPERCASE to avoid errors.

E

Horizon CNOS Connections

CNOS overview

SNA LU 6.2 supports single-session and parallel-session connections. A parallel-session LU can have multiple concurrently active sessions with a given partner LU. A single-session LU cannot activate another session until the current session is deactivated. SNA CNOS verbs dynamically control the number, activation, and deactivation of parallel sessions between two LUs.

CNOS has no meaning for the TCP/IP protocol, and therefore the features in this appendix are not applicable for connections using a TCP/IP protocol.

This section provides an overview of CNOS terminology and basic operation.

CNOS terminology

Each CNOS command has a ***source*** and a ***target***. The source LU is the LU that issues the CNOS request. The target LU is the LU that receives the CNOS request and issues the CNOS reply.

CNOS requests are issued only over ***control sessions***. There are two control sessions for each pair of partner LUs. These sessions are dedicated to regulating the remaining ***conversational sessions*** bound between the LU partners. A conversational session is a session used for conversations between user applications.

A CNOS command can apply to all sessions existing between two logical units or just those in a particular ***mode***. A mode is a subset of the sessions between two LUs. All sessions in a mode, by definition, have a common ***logmode*** name and common session characteristics.

A logmode names a SNA Communications Server (formerly VTAM) table entry that contains predefined session characteristics that SNA Communications Server uses to establish sessions between two logical units. The CNOS partners' common session characteristics include request unit size, access security, trusted logins, and support for LU 6.2 and the CNOS protocol.

A mode is identified to the LU partners by a ***mode name*** which must be unique to the LU it subdefines.

The ***session limit*** is the maximum number of concurrent sessions that can exist between two LUs using the same mode.

CNOS session control

The three basic CNOS verbs are INITIALIZE, RESET, and CHANGE:

- INITIALIZE_SESSION_LIMIT changes the session limit from zero to a nonzero number.
- RESET_SESSION_LIMIT changes the session limit to zero.
- CHANGE_SESSION_LIMIT increases or decreases the session limits

Setting session limits

When a CNOS INITIALIZE verb sets a session limit it specifies:

- Source requested maximum session count
- Guaranteed number of source contention winners
- Guaranteed number of target contention winners

Source contention winners means the number of sessions the source has reserved for itself. Target contention winners means the number of sessions the source is willing to reserve for the target.

The target may negotiate these values to meet its requirements and uses several negotiation rules to do so. The target replies to the source with values for the bulleted items above and will indicate whether the values returned have been negotiated.

Deactivating sessions

A CNOS RESET verb specifies:

- Whether the source or target is responsible for session deactivation.
- Whether the reset applies to one or all modes between the local and remote LU.

When one resets the session limit, one of the two LUs must deactivate the sessions between them to make the active session count equal to zero. The RESPONSIBLE parameter of RESET designates who is responsible for deactivating the required number of sessions.

Sessions are deactivated in an orderly manner:

- Conversations are allowed to finish.
- The responsible LU indicates that it will send no new conversation requests by sending a BIS (Bracket Initiation Stop) request.
- The target of this request acknowledges and concurs by sending a BIS reply.
- The responsible LU asks SNA Communications Server to deactivate the session by sending an UNBIND request.

Changing session limits

When a CNOS CHANGE verb increases or decreases session limits, it specifies the same parameters (session count and contention winners) as INITIALIZE. If sessions need to be deactivated to meet the new session limits, CHANGE (like RESET) specifies the node responsible for session deactivation and the modes affected.

CNOS minimum support

Horizon provides the CNOS minimum support set, which includes the components listed below:

- A control-operator transaction program that can issue the CNOS minimum support set verbs
- Presentation services for the control operator program
- Two control sessions for every CNOS LU pair
- The CNOS service transaction program

Minimum support verbs

The CNOS minimum support verbs include the control operator, LU definition, and display verbs shown in Table E-1. The table also shows the Model 204 commands (presentation) for those verbs:

Table E-1. CNOS verbs with equivalent Model 204 commands

CNOS verb	Meaning for Model 204	Model 204 command
INITIALIZE_SESSION_LIMIT (send support)	Horizon initiates CNOS communications with a partner LU	START SESSIONGROUP

Table E-1. CNOS verbs with equivalent Model 204 commands (Continued)

CNOS verb	Meaning for Model 204	Model 204 command
INITIALIZE_SESSION_LIMIT(receive support)	Horizon responds to a remote LU's CNOS_INITIALIZE request	—
RESET_SESSION_LIMIT (send support)	Horizon shuts down conversation traffic	STOP SESSIONGROUP
RESET_SESSION_LIMIT (receive support)	Horizon responds to a remote LU's CNOS RESET request	—
RESET_SESSION_LIMIT MODENAME(ALL)	Horizon lets all conversations for all sessiongroups with a remote to finish, unbinds the sessions, and frees the sessiongroups	STOP REMOTE
CHANGE_SESSION_LIMIT (receive support)	Horizon responds to a remote LU's CHANGE_SESSION_LIMITrequest	—
DEFINE_TP	Defines Horizon process	DEFINE PROCESS
DEFINE_LOCAL_LU	Defines Horizon link	DEFINE LINK
DEFINE_REMOTE_LU	Defines Horizon remote	DEFINE REMOTE
DEFINE_MODE	Defines Horizon sessiongroup	DEFINE SESSIONGROUP
DISPLAY_TP	Shows Horizon CNOS process usage	MONITOR PROCESS
DISPLAY_LOCAL_LU	Shows Horizon CNOS link usage	MONITOR LINK
DISPLAY_REMOTE_LU	Shows Horizon CNOS remote usage	MONITOR REMOTE
DISPLAY_MODE	Shows Horizon sessiongroup usage	MONITOR SESSIONGROUP

Optional verb

Model 204 also provides send support for CHANGE_SESSION_LIMIT, allowing Model 204 user to initiate execution of MODIFY SESSIONGROUP. This functionality is not required as part of the CNOS minimum support set.

Managing the Horizon CNOS network

Setting up Horizon CNOS support requires modifications to the basic Horizon network entity definitions, two CNOS-specific entity definitions, and two SNA Communications Server mode table entry definitions.

Horizon network control commands are modified to apply to CNOS entities and include some functionality not available to non-CNOS Horizon.

Defining CNOS network entities to Model 204

You need to define two CNOS-only network entities, and you need to follow the CNOS-specific syntax for the three basic Horizon network entities.

Remotes and sessiongroups

The following Model 204 intersystem entities are unique to and required for Horizon links with CNOS sessions:

Intersystem entity	Description
Remote	Identifies the remote LU and specifies the local link to which the remote node connects and the login security for the link.
Sessiongroup	A group of sessions with the same network properties (from a common remote and link) and the same session characteristics (from a common SNA Communications Server logmode entry).

You request CNOS support by including a remote entity and one or more sessiongroup entity definitions with your link, processgroup, and process definitions.

Links, processgroups, and processes

The basic Horizon entity definitions are documented on the Rocket Model 204 documentation wiki command pages:

http://m204wiki.rocketsoftware.com/index.php/Category:Commands

- DEFINE LINK command: Horizon for TCP/IP
- DEFINE LINK command: Horizon for VTAM
- DEFINE PROCESS command: Horizon for TCP/IP and VTAM
- DEFINE PROCESSGROUP command: Horizon for TCP/IP
- DEFINE PROCESSGROUP command: Horizon for VTAM

Defining CNOS support to SNA Communications Server

SNA Communications Server network definition for Horizon CNOS differs from non-CNOS only in the SNA Communications Server mode table entry requirements. Horizon CNOS connections require two mode entries:

- One entry for the characteristics of the CNOS control sessions
- One or more entries for the characteristics of the CNOS conversation sessions

Coding the SNA Communications Server APPL definition

Follow the directions in Chapter 3 for non-CNOS APPL definitions.

Coding the mode table entries

Define a mode entry whose entry name and LOGMODE parameter value is SNASVCMG. You can use the SNASVCMG entry in the default mode table supplied by IBM.

Define one or more mode entries for the CNOS conversation sessions. Make sure each entry has an entry name and LOGMODE parameter value that matches the value of the MODENAME parameter of a DEFINE SESSIONGROUP command.

For each conversation session entry, follow the parameter recommendations on page 41 for non-CNOS connections, with one exception:

```
Specify PSERVIC=X’060200000000000000102300’
```

Handling CNOS errors

Rejecting a CNOS connection

If you issue an inbound request for an existing process that does not seem (according to the definitions) to have a logical connection with the CNOS connection over which the connection request has come, Model 204 issues the following message:

```
M204.2837: NO LOGICAL CONNECTION BETWEEN PROCESS %C AND
REMOTEID %C
```

Typical CNOS connection errors

Typically, Model 204 informs you of errors involving the CNOS connection by issuing the 2260 error message that follows. The message suits a variety of error situations by substituting one of multiple qualifying phrases according to the error characteristics.

```
M204.2260: CONNECTION [INITIALIZATION
  | CHANGE OF SESSION LIMITS | RESET
  | RESET OF ALL MODENAMES]
  WITH [partner-LU] [COMPLETED SUCCESSFULLY
  | DID NOT COMPLETE]
  - [qualifier], SOURCE WAS [originating-LU]
```

Where

Argument	Means…
CONNECTION INITIALIZATION COMPLETED SUCCESSFULLY	Initial intersystem handshake is completed and user procedures can run over the connection.
CONNECTION CHANGE OF SESSION LIMITS COMPLETED SUCCESSFLLY	By request of one partner, the capacity of the intersystem connection was increased or decreased.
CONNECTION RESET COMPLETED SUCCESSFULLY	One partner is draining the conversational sessions between the two partners that belong to the relevant SESSIONGROUP. All sessions will be terminated when the drain completes. You must reinitialize the connection to begin another user conversation.
CONNECTION RESET OF ALL MODENAMES COMPLETED SUCCESSFULLY	One partner is draining all service and all conversational session for each of the SESSIONGROUPs between the two partners. All sessions will be terminated when the drain completes. You must reinitialize the connection to begin another user conversation.
partner-LU	Specifies the name of the conversation partner program. It is the same name used in the DEFINE PROCESS command for the server process at the remote node.
qualifier	One of the phrases explained in Table E-2 on page 183
originating-LU	Specifies the name of the source program. It is the same name used in the DEFINE PROCESS command for the local server.

Note: Periodically check that the connection is being set up and shut down as planned.

Table E-2. Message 2260 qualifiers

Qualifying phrase	Meaning
ACCEPT END / INIT END FAILED	An internal error has occurred.

Table E-2. Message 2260 qualifiers (Continued)

Qualifying phrase	Meaning
COMMAND RACE DETECTED	Another CNOS command was in progress when you submitted your START SESSIONGROUP or STOP SESSIONGROUP or STOP REMOTE. Please try again.
ERROR UNLOCKING SGRD	A serious internal error has occurred.
MODENAME IN USE BY A PGRD	The MODENAME value in your sessiongroup definition has already been allocated to a processgroup owned by the same link.
MODENAME IN USE BY AN SGRD	The MODENAME value in your sessiongroup definition has already been allocated to a processgroup owned by the same remote.
MODENAME NOT DEFINED	The target of the CNOS request could not find a sessiongroup and remote definition that matched the source's START SESSIONGROUP or STOP SESSIONGROUP or STOP REMOTE.
NEGOTIATED	The target negotiated the source's CNOS request.
NOT NEGOTIATED	The target accepted the CNOS request as it was and did not attempt to change it.
OPEN PROCESS FAILED	An internal error has occurred.
PROTOCOL VIOLATION DETECTED	The CNOS source or target has committed a protocol error (for example, requesting more winners than maximum sessions, not specifying a mode name on a set or reset command, trying to negotiate a reset command, etc.). See "Parallel Session Support" in the *SNA Format and Protocol Reference*.
RECEIVE FAILED	An internal error has occurred.
REMOTE NOT DEFINED	The named remote has no remote entity definition.
REMOTEID IN USE BY A PGRD	The REMOTEID specified in the remote entity definition was previously allocated to a non-CNOS processgroup. A non-CNOS processgroup may not share a REMOTEID with a remote unless they are owned by different links.
REMOTEID IN USE BY AN RMTD	The REMOTEID specified in the remote entity definition was previously allocated to another remote owned by the same link.
RMTD CHAINING ERROR	A serious internal error has occurred.
RMTD LINK NOT DEFINED	Model 204 is unable to find an entity definition for the link named in the remote entity definition.
RMTD NOT INIT GCORE FAILED	Model 204 is unable to get the storage necessary to initialize a remote control block.
RMTD NOT INIT RESET RQ RECVD	A STOP REMOTE has been received for a remote that was already stopped or was never initialized (none of its sessiongroups was ever started)

Table E-2. Message 2260 qualifiers (Continued)

Qualifying phrase	Meaning
SEE PREVIOUS MESSAGE	A bug has occurred. See the previous bug message.
SEND FAILED	An internal error has occurred.
SESSION LIMIT AT 0	The target has refused the source's request because its session limit is already zero.
SESSIONGROUP ALREADY STOPPED	Model 204 received a STOP SESSIONGROUP request for a sessiongroup that was already stopped (by a previous STOP SESSIONGROUP or STOP REMOTE).
SESSIONGROUP NOT DEFINED	A. The named sessiongroup has no entity definition. B. The named sessiongroup has no corresponding definition at the remote node that matches the sessiongroup's REMOTEID and MODENAME values.
SGRD CHAINING ERROR	A serious internal error has occurred.
SGRD IN LOCK DENIED STATE	A command race occurred and your request was the loser. Please try again.
SGRD LOCK BY SOURCE	A command race occurred and your request was the loser. Please try again.
SGRD LOCKED BY TARGET	A command race occurred and your request was the loser. Please try again.
SGRD NOT INIT GCORE FAILURE	Model 204 is unable to allocate the storage needed to initialize the SGRD.
SNASVCMG CHAINING ERROR	A serious internal error has occurred.
SNASVCMG IN USE BY A PGRD	A serious internal error has occurred.
SNASVCMG NOT DEFINED	A serious internal error has occurred.
SNASVCMG NOT INIT- GCORE FAIL	Model 204 is unable to allocate the storage needed to initialize the SGRD for SNASVCMG.
SNASVCMG NOT INIT- RESET RECV	A serious internal error has occurred.

Index