Tivoli NetView for OS/390

Bridge Implementation

Version 1 Release 4

SC31-8238-03

Tivoli NetView for OS/390

Bridge Implementation

Version 1 Release 4

SC31-8238-03

Tivoli NetView for OS/390 Bridge Implementation

Trademarks

IBM, the IBM logo, Tivoli, the Tivoli logo, AIX, AnyNet, APPN, AT, C/370, CICS, DB2, IMS, KnowledgeTool, Language Environment, LPDA, MVS/ESA, NetFinity, NetView, OPC, OS/2, OS/390, RACF, RS/6000, Tivoli Global Enterprise Manager, Tivoli Enterprise Console, Tivoli Management Framework, TME 10, VM/ESA, VSE/ESA, and VTAM are trademarks or registered trademarks of International Business Machines Corporation or Tivoli Systems Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Notices

References in this publication to Tivoli Systems or IBM products, programs, or services do not imply that they will be available in all countries in which Tivoli Systems or IBM operates. Any reference to these products, programs, or services is not intended to imply that only Tivoli Systems or IBM products, programs, or services can be used. Subject to valid intellectual property or other legally protectable right of Tivoli Systems or IBM, any functionally equivalent product, program, or service can be used instead of the referenced product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by Tivoli Systems or IBM, are the responsibility of the user. Tivoli Systems or IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, New York 10504-1785, U.S.A.

Programming Interfaces

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain services of Tivoli NetView for OS/390.

Contents

Preface

This book explains how to plan for and implement the NetView® Bridge function of the NetView program. This includes accessing NetView Bridge remotely.

NetView Bridge is a set of application programming interfaces (APIs) that allows the NetView program to interact with various types of databases in the MVS environment when combined with a database server, such as Information/System-NetView Bridge Adapter. You can use NetView Bridge to write automation applications that can access data stored in your MVS database and can transport data between the NetView program and your database.

In addition to accessing your database from the MVS environment, NetView Bridge can remotely access your MVS database from the MVS environment. To remotely access your database, you must install NetView Bridge on the host MVS system where your database resides and install NetView Bridge remote access functions on your remote NetView system.

Step-by-step instructions tell you how to set up and customize NetView Bridge and NetView Bridge remote access for your environment. After you complete these instructions, your NetView Bridge and NetView Bridge remote access functions will be operational.

Who Should Read This Document

System programmers and NetView operators should use this book to set up and customize the NetView Bridge and NetView Bridge remote access functions. Use this document when you are installing the NetView program and when you code applications that use the functions of NetView Bridge.

As a programmer or NetView operator, you should be familiar with the MVS operating system and your target database. If you access NetView Bridge remotely, you should also be familiar with the remote operating system. If you have purchased a database server, such as Information/System-NetView Bridge Adapter, you should be familiar with how to install the database server.

In addition, you should know how to create and alter job control language (JCL) statements. It is helpful to have a thorough understanding of one of the following: the Restructured Extended Executor (REXX) language, the NetView command list language, PL/I, or C.

Prerequisite and Related Documents

To read about the new functions offered in this release, refer to the *Tivoli NetView for OS/390 Installation: Getting Started*.

You can find additional product information on these Internet sites:

Table 1. Resource Address (URL)

IBM	http://www.ibm.com/
Tivoli Systems	http://www.tivoli.com/
Tivoli NetView for OS/390	http://www.tivoli.com/nv390

The Tivoli NetView for OS/390 home page offers demonstrations of NetView, related products, and several free NetView applications you can download. These applications can help you with tasks such as:

- Getting statistics for your automation table and merging the statistics with a listing of the automation table
- Displaying the status of a JES job or cancelling a specified JES job
- Sending alerts to NetView using the program-to-program interface (PPI)
- Sending and receiving MVS commands using the PPI
- Sending TSO commands and receiving responses

What This Document Contains

This document provides an overview of NetView Bridge and NetView Bridge remote access. It describes how to set up, create a database, and write a command procedure. This document also describes how to use NetView Bridge, optimize it for best performance, and explains how to report any problems you might have. The appendixes include special information that mitht prove helpful when using NetView Bridge.

Conventions Used in This Document

The document uses several typeface conventions for special terms and actions. These conventions have the following meaning:

Bold	Commands, keywords, flags, and other information that you must use literally appear like **this**, in **bold**.
Italics	Variables and new terms appear like *this*, in *italics*. Words and phrases that are emphasized also appear like *this*, in *italics*.
`Monospace`	Code examples, output, and system messages appear like `this`, in a `monospace` font.
ALL CAPS	Tivoli NetView for OS/390 commands are in ALL CAPITAL letters.

Platform-specific Information

For more information about the hardware and software requirements for NetView components, refer to the *Tivoli Netview for OS/390 Licensed Program Specification.*

Terminology

For a list of Tivoli NetView for OS/390 terms and definitions, refer to http://www.networking.ibm.com/nsg/nsgmain.htm.

For brevity and readability, the following terms are used in this document:

NetView

- Tivoli NetView for OS/390 Version 1 Release 4
- Tivoli NetView for OS/390 Version 1 Release 3
- TME 10 NetView for OS/390 Version 1 Release 2
- TME 10 NetView for OS/390 Version 1 Release 1
- IBM NetView for MVS Version 3
- IBM NetView for MVS Version 2 Release 4
- IBM NetView Version 2 Release 3

MVS MVS/ESA, OS/390, or z/OS operating systems.

Tivoli Enterprise software
Tivoli software that manages large business networks.

Tivoli environment
The Tivoli applications, based upon the Tivoli Management Framework, that are installed at a specific customer location and that address network computing management issues across many platforms. In a Tivoli environment, a system administrator can distribute software, manage user configurations, change access privileges, automate operations, monitor resources, and schedule jobs. You may have used TME 10 environment in the past.

TME 10
In most product names, TME 10 has been changed to Tivoli.

V and R
Specifies the version and release.

VTAM and TCP/IP
VTAM and TCP/IP for OS/390 are included in the IBM Communications Server for OS/390 element of the OS/390 operating system. Refer to http://www.software.ibm.com/enetwork/commserver/about/csos390.html.

Unless otherwise indicated, references to programs indicate the latest version and release of the programs. If only a version is indicated, the reference is to all releases within that version.

When a reference is made about using a personal computer or workstation, any programmable workstation can be used.

Reading Syntax Diagrams

Syntax diagrams start with double arrowheads on the left (►►) and move along the main line until they end with two arrowheads facing each other (►◄).

As shown in the following table, syntax diagrams use ***position*** to indicate the required, optional, and default values for keywords, variables, and operands.

Table 2. How the Position of Syntax Diagram Elements Is Used

Element Position	Meaning
On the command line	Required
Above the command line	Default
Below the command line	Optional

Required Syntax

The command name, required keywords, variables, and operands are always on the main syntax line. Figure 1 on page x specifies that the *resname* variable must be used for the CCPLOADF command.

CCPLOADF

Figure 1. Required Syntax Elements

Keywords and operands are written in uppercase letters. Lowercase letters indicate variables such as values or names that you supply. In Figure 2, MEMBER is an operand and *membername* is a variable that defines the name of the data set member for that operand.

TRANSMSG

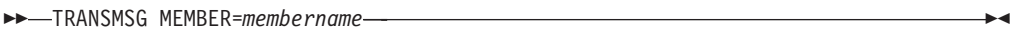


Figure 2. Syntax for Variables

Optional Keywords and Variables

Optional keywords, variables, and operands are below the main syntax line. Figure 3 specifies that the ID operand can be used for the DISPREG command, but is not required.

DISPREG

Figure 3. Optional Syntax Elements

Default Values

Default values are above the main syntax line. If the default is a keyword, it appears only above the main line. You can specify this keyword or allow it to default.

If an operand has a default value, the operand appears both above and below the main line. A value below the main line indicates that if you choose to specify the operand, you must also specify either the default value or another value shown. If you do not specify an operand, the default value above the main line is used.

Figure 4 on page xi shows the default keyword STEP above the main line and the rest of the optional keywords below the main line. It also shows the default values for operands `MODNAME=*` and `OPTION=*` above and below the main line.

RID

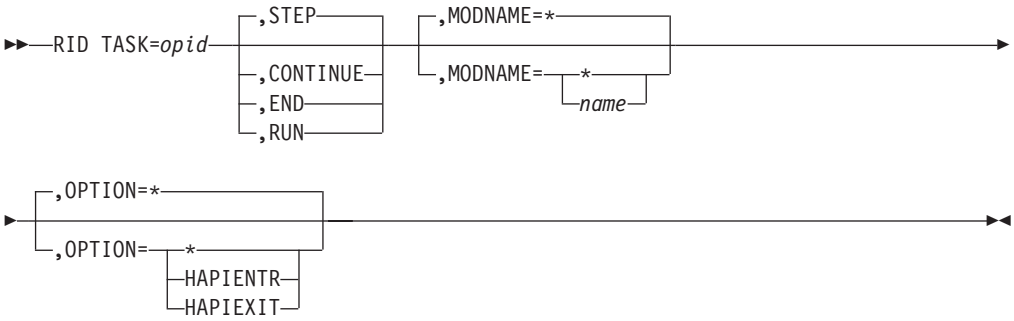


Figure 4. Sample of Defaults Syntax

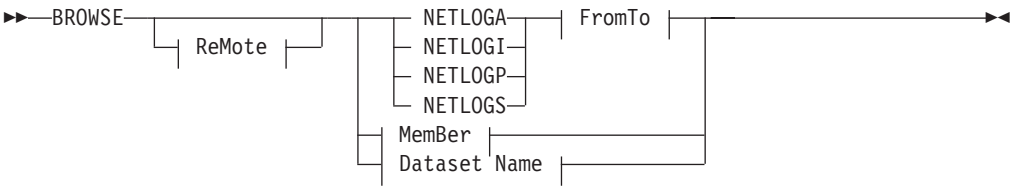
Long Syntax Diagrams

When more than one line is needed for a syntax diagram, the continued lines end with a single arrowhead (►). The following lines begin with a single arrowhead (►), as shown in Figure 4.

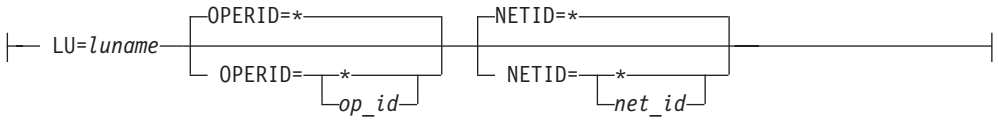
Syntax Fragments

Commands that contain lengthy groups or a section that is used more than once in a command are shown as separate fragments following the main diagram. The fragment name is shown in mixed case. See Figure 5 on page xii for a syntax with the fragments `ReMote` and `FromTo`.

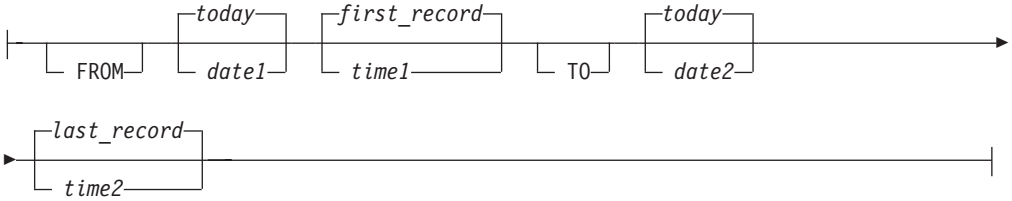
BROWSE



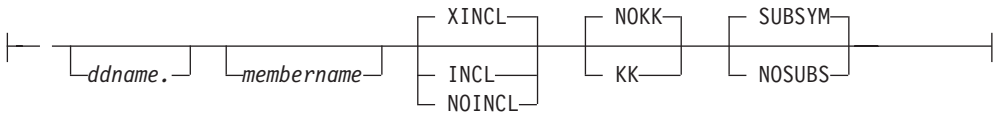
ReMote:



FromTo:



MemBer:



Dataset Name:

Figure 5. Sample Syntax Diagram with Fragments

Commas and Parentheses

Required commas and parentheses are included in the syntax diagram. When an operand has more than one value, the values are typically enclosed in parentheses and separated by commas. In Figure 6 on page xiii, the OP operand, for example, contains commas to indicate that you can specify multiple values for the *testop* variable.

CSCF

PurgeBefore

Pu

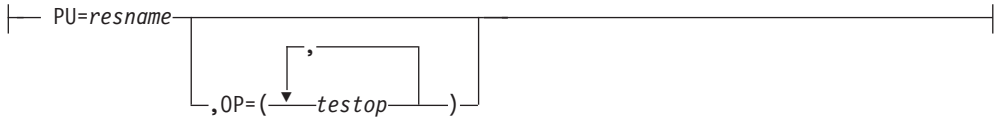


Figure 6. Sample Syntax Diagram with Commas

If a command requires positional commas to separate keywords and variables, the commas are shown before the keyword or variable, as in Figure 4 on page xi.

For example, to specify the BOSESS command with the *sessid* variable, enter:

```
NCCF BOSESS applid,,sessid
```

You do not need to specify the trailing positional commas. Positional and non-positional trailing commas either are ignored or cause the command to be rejected. Restrictions for each command state whether trailing commas cause the command to be rejected.

Highlighting, Brackets, and Braces

Syntax diagrams do not rely on highlighting, underscoring, brackets, or braces; variables are shown italicized in hardcopy or in a differentiating color for NetView help and BookManager online books.

In parameter descriptions, the appearance of syntax elements in a diagram immediately tells you the type of element. See Table 3 for the appearance of syntax elements.

Table 3. Syntax Elements Examples

This element...	Looks like this...
Keyword	CCPLOADF
Variable	*resname*
Operand	MEMBER=*membername*
Default	*today* or INCL

Abbreviations

Command and keyword abbreviations are described in synonym tables after each command description.

Accessing Publications Online

The Tivoli Customer Support Web site (http://www.tivoli.com/support/) offers a guide to support services (the *Customer Support Handbook*); frequently asked questions (FAQs); and technical information, including release notes, user's guides, redbooks, and white papers. You can access Tivoli publications online at http://www.tivoli.com/support/documents/. The documentation for some products is available in PDF and HTML formats. Translated documents are also available for some products.

To access most of the documentation, you need an ID and a password. To obtain an ID for use on the support Web site, go to http://www.tivoli.com/support/getting/.

Resellers should refer to http://www.tivoli.com/support/smb/index.html for more information about obtaining Tivoli technical documentation and support.

Business Partners should refer to "Ordering Publications" for more information about obtaining Tivoli technical documentation.

Note: Additional support is also available on the NETVIEW CFORUM (Customer Forum) through the IBMLink system. This forum is monitored by NetView developers who answer questions and provide guidance. When a problem with the code is found, you are asked to open an official problem management record (PMR) to get resolution.

Ordering Publications

Order Tivoli publications online at http://www.tivoli.com/support/Prodman/html/pub_order.html or by calling one of the following telephone numbers:

- U.S. customers: (800) 879-2755
- Canadian customers: (800) 426-4968

Providing Feedback about Publications

We are very interested in hearing about your experience with Tivoli products and documentation, and we welcome your suggestions for improvements. If you have comments or suggestions about our products and documentation, contact us in one of the following ways:

- Send e-mail to pubs@tivoli.com.
- Fill out our customer feedback survey at http://www.tivoli.com/support/survey/.

Contacting Customer Support

The *Tivoli Customer Support Handbook* at http://www.tivoli.com/support/handbook/ provides information about all aspects of Tivoli Customer Support, including the following:

- Registration and eligibility
- How to contact support, depending on the severity of your problem
- Telephone numbers and e-mail addresses, depending on the country you are in
- What information you should gather before contacting support

Chapter 1. Introduction to NetView Bridge and NetView Bridge Remote Access

This chapter describes the functions and requirements for NetView Bridge and NetView Bridge remote access. NetView Bridge provides a set of application programming interfaces (APIs) that gives you an effective means of connecting an MVS NetView program to target databases or transaction processors. You can also access target databases from a remote MVS, VM, or VSE system through the resident NetView program running in MVS, using the NetView high performance transport.

Understanding NetView Bridge

The NetView program collects data from the resources in your network. This data includes information about session awareness, system configuration, and problems within your network.

NetView Bridge can help track information concerning your resources, network, or configuration by creating and updating records in a non-NetView database. This database must reside on an MVS system.

The NetView program collects data that might make updates necessary to your configuration database. NetView Bridge allows you to write applications that can access data stored in a configuration database and transport data between the NetView program and your database. These programs can update your database as often as necessary.

You can update data and transfer data to your database by using the NetView Bridge Requester API and the server support API. These functions use high-level language (HLL) programs and user-written command lists to perform your work. You can invoke target database functions that are not in the NetView address space and develop your own transaction processors.

Understanding NetView Bridge Remote Access

NetView Bridge uses the NetView Bridge Requester API to access servers that reside on hosts other than the one in which the transaction was generated. You can collect data from the resources in your network and create and update records in a database residing in an MVS system. Using the remote dispatcher, a NetView program running on a remote MVS, VM, or VSE operating system can access these records.

Transactions are forwarded to the resident host, which is an MVS system running NetView Bridge, by using the NetView high performance transport API, which allows applications in two network nodes to communicate over an LU 6.2 session.

How NetView Bridge Functions

NetView Bridge uses APIs, a bridge dispatcher, servers, and transaction processors that must be installed in your host. The primary components of NetView Bridge are shown in Figure 7 on page 3.

To access your database, a transaction (the transaction request and the data required to process it) should flow in the following manner:

1. A user-written command procedure (NetView command list language, REXX, or HLL program) uses the NetView Bridge Requester API to request that a transaction be sent to a database server. Examples of user-written command procedures and information about how to write these procedures are in “Chapter 4. Writing a NetView Bridge Command Procedure Using the NetView Bridge Requestor API” on page 55.
2. The bridge dispatcher passes this request to the input queue of an available database server in the program-to-program interface.
3. A database server, such as the Information/System-NetView Bridge Adapter, uses the server support API to execute a transaction. The server performs the requested transaction, such as IBUPDATE of the Information/System-NetView Bridge Adapter, by invoking the appropriate transaction processor. This transaction processor works directly with the target database.
4. If the transaction processor creates a reply for this request, it uses the server support API to pass the reply to the bridge dispatcher through the output queue in the program-to-program interface. The bridge dispatcher then returns the reply to a user-written command procedure in the form of a message.
5. A user-written command procedure uses the NetView Bridge Requester API, specifically the TRANRCV command or the CNMGETP service routine, to extract the reply data from the message.

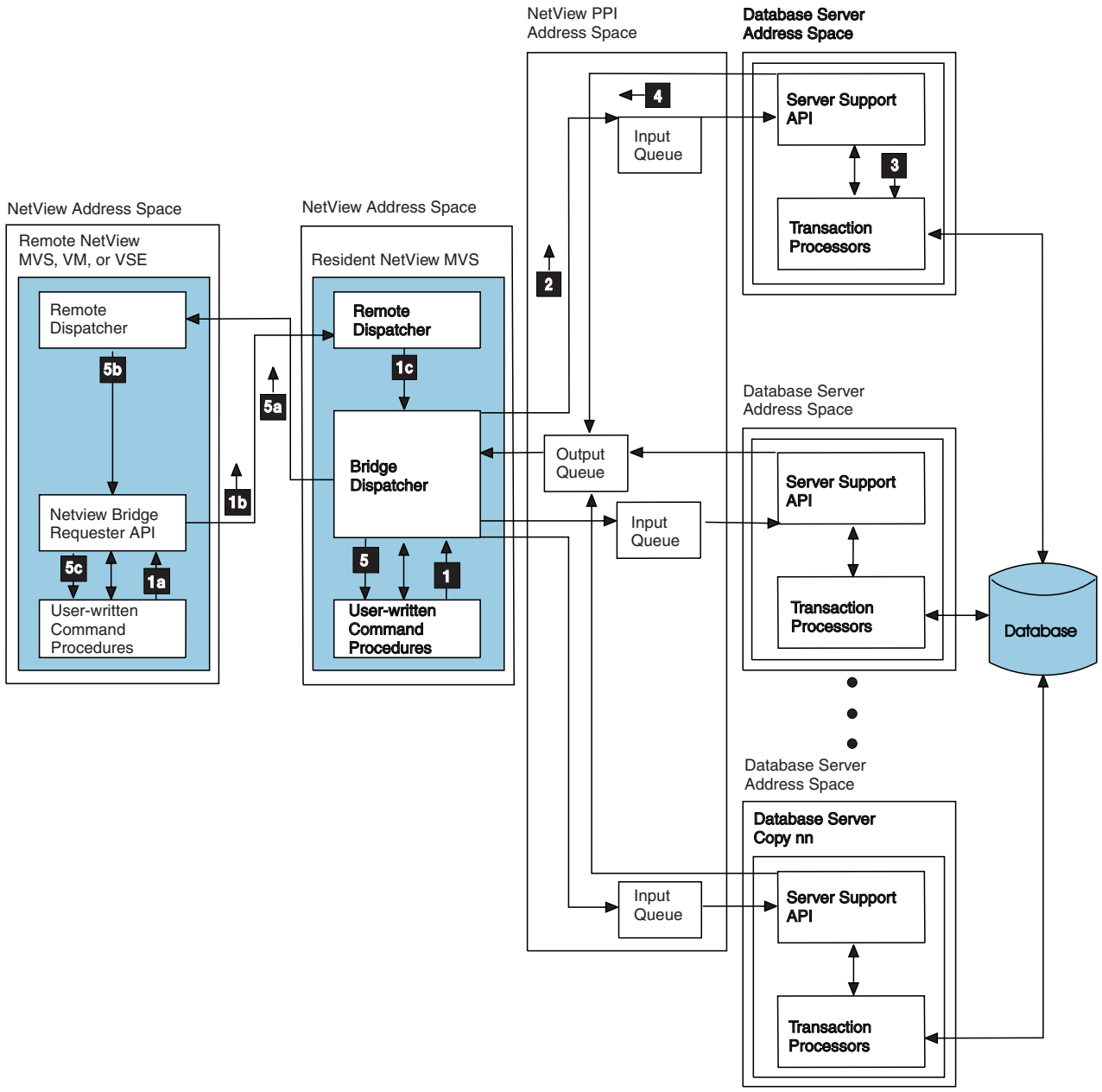


Figure 7. Primary Components of NetView Bridge

How NetView Bridge Remote Access Functions

NetView Bridge remote access functions allow you to access your MVS database from a remote NetView program by sending requests to a resident NetView Bridge operating in an MVS system. The primary components of NetView Bridge remote access are shown in Figure 7. NetView Bridge uses user-written command procedures, APIs, the remote dispatcher, servers, the program-to-program interface, and transaction processors that must be installed in your host for remote access.

For remote access to your database, a transaction should flow in the following manner:

1a A user-written command procedure (NetView command list language, REXX, or HLL program in an MVS system) uses the NetView Bridge Requester API to request that a transaction be sent to the remote dispatcher. Examples of user-written command procedures and information about how to write these procedures are in “Chapter 4. Writing a NetView Bridge Command Procedure Using the NetView Bridge Requestor API” on page 55.

1b The NetView high performance transport API (DSIHPDST) acts as an interface between the remote NetView system and the resident NetView system running in MVS to send the request to the remote dispatcher in the resident NetView system.

1c The remote dispatcher sends the request to the bridge dispatcher running in the resident NetView system.

2 The bridge dispatcher passes this request to the input queue of an available database server in the program-to-program interface.

3 A database server, such as the Information/System-NetView Bridge Adapter, uses the server support API to retrieve the transaction. The server performs the requested transaction by invoking the appropriate transaction processor. This transaction processor works directly with the target database.

4 If the transaction processor creates a reply for this request, it uses the server support API to pass the reply to the bridge dispatcher through the output queue in the program-to-program interface.

5a The bridge dispatcher uses the NetView high performance transport API to send the reply to the remote dispatcher on the remote NetView system.

5b The remote dispatcher sends the reply to the user-written command procedure in the form of a message.

5c A user-written command procedure uses the NetView Bridge Requester API, specifically the TRANRCV command or the CNMGETP service routine, to extract the reply data from the message.

Components of NetView Bridge

You need a database server, a server support API, user-written command procedures, a NetView Bridge Requester API, a bridge dispatcher, and transaction processors to enable NetView Bridge to work. If the NetView program is running on a remote operating system, the remote bridge dispatcher is required. NetView Bridge also uses the NetView subsystem address space as the program-to-program interface. The following are descriptions of these components.

Database Server

A database server connects the NetView program and the target database. A database server consists of one or more transaction processors that run in an address space outside of the NetView program.

When a database server is started, an address space is created and the server support API is initialized. The server waits for a transaction to arrive in its server-defined input queue. When a transaction is received, the server invokes the appropriate transaction processor to perform the work.

If the transaction processor generates a response, the database server sends that response back to the NetView program using the server support API to send to the output queue, which the bridge dispatcher monitors. The database server then waits for another transaction.

When you request termination of the database server, the database server terminates the server support API and the address space.

Multiple servers allow you to process multiple transactions in parallel. Adding additional servers can help you handle increased throughput as your transaction volume increases, but you should expect performance trade-offs. Consult your database server documentation for instructions on how to create more copies of a server.

You can purchase a server for certain databases, such as the Information/System-NetView Bridge Adapter for the Information/Management database from IBM®, or you can develop one yourself using the server support API. See “Chapter 3. Creating a Database Server” on page 15 for more information on creating a database server using the server support API.

NetView Server Support API

The NetView server support API provides services to your database server. Your database server uses the server support API to initialize the API, to receive transactions, to send responses back to the NetView program, and to terminate the API.

User-Written Command Procedures

You use the NetView Bridge commands in your automation applications. To access a server from the NetView program, you must invoke NetView Bridge commands using the NetView command list language, REXX, or a high-level language (HLL) in an MVS system. Examples of user-written command procedures and information about how to write these procedures are in “Chapter 4. Writing a NetView Bridge Command Procedure Using the NetView Bridge Requestor API” on page 55.

NetView Bridge Requester API

The NetView Bridge Requester API handles the commands in your command procedures, passing transaction requests to the transaction processors. The NetView Bridge Requester API returns a response to your command procedures (if one is expected) and must be invoked to get the reply data.

Bridge Dispatcher

A NetView bridge dispatcher runs as an autotask, which is a task that does not require a terminal or a logged-on user. This dispatcher is present only in the MVS host on which the target database resides.

A bridge dispatcher executes the NetView Bridge command RTRINIT to establish the NetView Bridge environment and pass transactions between user-written command procedures and database servers. The bridge dispatcher places transactions in the input queue in the program-to-program interface for an available database server.

The RTRINIT command is driven by the profile of a NetView autotask. This autotask serves as the interface to a specific set of database servers and defines three user-provided queue names to the program-to-program interface. These queues are

the output queue, to which the database servers send all their output, the ready queue, which contains ready tokens indicating which database servers are available, and the hold queue, which holds transaction requests when no database servers are available. For information on the program-to-program interface, refer to "Understanding the NetView Program-to-Program Interface" in the *Tivoli NetView for OS/390 Application Programmer's Guide*.

As shown in Figure 7 on page 3, more than one copy of a server can interface with a bridge dispatcher. One or more server copies per bridge dispatcher are collectively referred to as a server set. If your server set consists of more than one server, each copy of the server must be functionally identical.

You can define more than one bridge dispatcher to your NetView system by defining multiple autotasks. Each bridge dispatcher can have one or more copies of a server assigned to it. Each server set sends output to a single output queue.

The bridge dispatcher can also communicate with a remote bridge dispatcher by using the NetView high performance transport API (DSIHPDST).

Transaction Processors

A database server can contain one or more transaction processors. These transaction processors interact directly with the database. The effect of transactions and the use of the fields within them depend upon the transaction processor and how it works with the database. Only the format of transactions is defined by NetView Bridge.

Remote Bridge Dispatchers (Remote Access Only)

A remote bridge dispatcher runs as a NetView autotask and executes the REMOTEBR command to register the remote dispatcher with the NetView high performance transport (DSIHPDST). The dispatcher enables transactions to be passed between user-written command procedures on the remote host and the MVS host on which the target database resides.

Two remote dispatchers are required for remote function of NetView Bridge: one in the MVS host on which the target database resides, and one on the remote MVS, VM, or VSE system.

NetView Subsystem Address Space

The NetView subsystem address space contains the program-to-program interface queues used by NetView Bridge. You must recycle the NetView Bridge dispatcher autotask and the database server every time a new program-to-program interface is activated.

Chapter 2. Setting up NetView Bridge and NetView Bridge Remote Access

This chapter contains the steps for setting up NetView Bridge and NetView Bridge remote access. An activity checklist is included to help you track your progress.

Before you proceed with this chapter, your system and the NetView program installation must meet certain criteria:

- The NetView program must be installed according to the installation guides included with the product. If you intend to use one of the high-level languages (HLLs) to create NetView Bridge command procedures, you must have completed the procedures for using high-level languages with the NetView program.
- You must have a database server, such as Information/System-NetView Bridge Adapter, or write your own database server.
- Your target database must reside on the same system as NetView Bridge.
- If you are using remote access, a remote dispatcher must be running on both your target NetView (MVS) system and on your remote NetView (MVS, VM, or VSE) system. The NetView Bridge dispatcher must be running on your MVS target NetView system.

Activity Checklist

Successfully setting up NetView Bridge and NetView Bridge remote access involves several activities. The following is an overview of the activities that you perform in subsequent chapters in this book. You can use this overview as a checklist.

1. Perform the steps in this chapter (beginning with “Preparing to Set Up NetView Bridge” on page 8) to set up and customize NetView Bridge. Follow each step in order, completing each step before proceeding to the next.
2. Perform the steps in this chapter (beginning with “Preparing to Set Up Remote Access to NetView Bridge” on page 11) to set up NetView Bridge remote access. Follow each step in order, completing each step before proceeding to the next.
3. If you intend to create your own database server, follow the instructions in “Chapter 3. Creating a Database Server” on page 15 to create the appropriate interface. If you have a database server, such as Information/System-NetView Bridge Adapter, see the documentation provided with the database server for information on how to install your server.
4. Follow the instructions in “Chapter 4. Writing a NetView Bridge Command Procedure Using the NetView Bridge Requestor API” on page 55 to write command procedures for using the NetView Bridge Requester API.
5. Follow the steps in “Chapter 5. Starting NetView Bridge, a Database Server, and NetView Bridge Remote Access” on page 79 to start NetView Bridge in your environment.
6. Optimize the performance of NetView Bridge. Some performance considerations are outlined in “Chapter 6. Adjusting NetView Bridge to Optimize Performance” on page 83. Other performance issues are probably unique to your installation.

Software Requirements

The following are the software requirements for NetView Bridge (local or remote access):

- VTAM® Version 4 Release 3 or a later release installed in a resident MVS host
- TME® 10 NetView for OS/390® Version 1 Release 1 (program number 5697-B82), or a later release

For VM and VSE you need the following:

- VTAM Version 3 Release 3 or a later release
- NetView Version 2 Release 3 running under VM/ESA® (program number 5756-051), or VSE/ESA™ (program number 5686-038)

Preparing to Set Up NetView Bridge

You might have multiple installations of the NetView program on your system. Regardless of the system they are installed on, any of the NetView programs are eligible to interact with your database through NetView Bridge installed on your MVS NetView system. You need to choose one of your MVS NetView systems on which to activate a NetView Bridge dispatcher and its associated server set. This is your resident NetView system. Once you have made this choice, you are ready to proceed with the setup steps on the following pages.

Note: Do not bypass any of the steps in this section. Ensure that you complete each task within each step (in order) before proceeding to the next step.

Step 1. Adding an Operator ID for the Dispatcher Autotask

The DSIOPF member of DSIPARM contains an operator ID, BRIGOPER, that you can customize with your own unique operator ID and profile name (*prfname*). This ID serves as the NetView interface to the database servers and is the destination task for NetView Bridge Requester API requests.

Note: Your bridge dispatcher autotasks should be exclusive to NetView Bridge.

To define the ID, you must modify the DSIOPF member of DSIPARM. For example, in the following OPERATOR statement, replace the highlighted name (BRIGOPER) with the unique operator ID that you have selected, and replace *prfname* with the name of the profile that you have selected.

```
BRIGOPER OPERATOR PROFILEN=prfname
```

This change takes effect the next time you recycle the NetView program.

Step 2. Creating a NetView Bridge Dispatcher Profile

You must create a new member in the NetView data set DSIPRF with the same name (*prfname*) that you selected in Step 1. This member authorizes the operator to act as a NetView Bridge dispatcher autotask. This profile specifies the NetView Bridge command RTRINIT as the initial command that runs at task initialization time to start NetView Bridge.

Using the NetView-supplied sample DSIPROFE (CNMS1051), you can create a new profile by changing the existing PROFILE statement and naming it to the new profile name (*prfname*).

The syntax for this profile is:

prfname PROFILE IC=RTRINIT *hqueue,rqueue,oqueue,hqueuel*

Notes:

1. For an explanation of the RTRINIT command and its syntax, see “Appendix A. NetView Bridge Initialization Commands” on page 87.
2. Replace *prfname* with the profile name that you defined in Step 1.
3. If you intend to use more than one autotask to communicate with the database servers, the positional queue names (*hqueue*, *rqueue*, *oqueue*) must be unique.

Step 3. Adding NetView Bridge Command Model Statements to the NetView Program

You must define the NetView Bridge commands to the NetView program by using command model (CMDMDL) statements. In MVS, these statements are in the DSICMD member of the DSIPARM data set. The following CMDMDL statements are required on the resident NetView system:

```
RTRINIT    CMDMDL   MOD=DSINBINT,TYPE=R,RES=N
RTRQUEUE   CMDMDL   MOD=DSINBQUE,TYPE=R,RES=Y
TRANRCV    CMDMDL   MOD=DSINBRCV,TYPE=R,RES=Y
TRANSND    CMDMDL   MOD=DSINBSND,TYPE=R,RES=Y
DSINBRSM   CMDMDL   MOD=DSINBRSM,TYPE=R,RES=Y
DSINBTRM   CMDMDL   MOD=DSINBTRM,TYPE=R,RES=Y
```

Note: If the NetView Bridge dispatcher is infrequently used, consider changing RES=Y to RES=N to save storage.

The RTRINIT command is described in “Appendix A. NetView Bridge Initialization Commands” on page 87, and the TRANRCV and TRANSND commands are described in “Appendix B. NetView Bridge Requester API Commands and Service Routines” on page 91. Commands RTRQUEUE, DSINBRSM, and DSINBTRM are internal NetView commands, and are not described.

Step 4. Creating a Server to Work with the NetView Program and Your Target Database

If you have purchased a database server that allows you to create your own transactions, see the documentation provided with that database server and then proceed to “Step 5. Creating a New Transaction Processor for an Existing Server” on page 10.

If you have not purchased a database server, develop your server software based on the instructions given in “Chapter 3. Creating a Database Server” on page 15. As part of creating your own server, you complete the tasks outlined in “Step 5. Creating a New Transaction Processor for an Existing Server” on page 10. Therefore, after completing the instructions in “Chapter 3. Creating a Database Server” on page 15, you can resume these procedures beginning with “Step 6. Creating NetView Command Procedures” on page 10.

Creating a new database server requires some programming effort. You must have an understanding of the server support API, and knowledge about accessing your target database. You can use either the PL/I or C programming language.

Step 5. Creating a New Transaction Processor for an Existing Server

At this point, you need to decide whether you want to add new transactions to your database server. If you have decided not to create any new transactions, proceed to "Step 6. Creating NetView Command Procedures".

Creating a transaction processor requires some programming effort. You must understand the server support API, how to access your database server, and how to access your target database. You can use either the PL/I or C programming language.

See "Creating a Transaction Processor" on page 20 for information on creating a transaction processor.

Step 6. Creating NetView Command Procedures

For this step, you need to do some programming in the language of your choice. The NetView Bridge Requester API supports the following languages:

- C
- NetView command list language
- PL/I
- REXX

Using the criteria outlined in "Designing Functions" in the *Tivoli NetView for OS/390 Customization Guide*, decide which programming language is most suitable for your needs. See "Chapter 4. Writing a NetView Bridge Command Procedure Using the NetView Bridge Requestor API" on page 55 for instructions on writing command procedures.

Step 7. Determining Whether to Use Multiple Copies of a Database Server

If you anticipate a large amount of traffic crossing the bridge to a given database, you might want to employ multiple database servers to handle the workload. Multiple database servers distribute the load and can reduce the time it takes to process requests and replies.

To determine whether adding multiple database servers is the best way for you to improve performance, see "Chapter 6. Adjusting NetView Bridge to Optimize Performance" on page 83.

The procedure for creating multiple database servers depends upon the type of database servers you already have. Consult the documentation for your particular server for instructions on how to create multiple database servers.

NetView Bridge requires minimal changes to accommodate additional servers. You must review queue sizes specified with the RTRINIT command in "Step 2. Creating a NetView Bridge Dispatcher Profile" on page 8 to ensure that you remain within your storage limitations. For additional details on setting queue limits, see "Adjusting Queue Limits" on page 84.

Step 8. Assigning Queue Names to Database Servers

You must assign a unique program-to-program interface queue name for each copy of a server. This queue is used by the server to receive input from the NetView bridge dispatcher. Consult the documentation for your particular server for detailed instructions on how to identify this name to the server.

Completing the Initial Setup of NetView Bridge

When you complete the setup of NetView Bridge, you can proceed to “Chapter 5. Starting NetView Bridge, a Database Server, and NetView Bridge Remote Access” on page 79 to ensure that you made all the necessary adjustments in setting up NetView Bridge.

Preparing to Set Up Remote Access to NetView Bridge

The NetView Bridge remote dispatcher can be defined on any NetView program operating in an MVS, VM, or VSE environment. However, the NetView Bridge remote dispatcher must also be defined on the same MVS NetView system in which NetView Bridge is operating. Once you choose the NetView programs on which to activate the NetView Bridge remote dispatcher, you are ready to proceed with the setup steps on the following pages.

Note: Do not bypass any of the steps in this section. Ensure that you complete each task within each step for all systems that will access NetView Bridge remotely.

Step 1. Adding an Operator ID for the Remote Dispatcher Autotask

An operator ID for the NetView Bridge remote dispatcher autotask must be defined on both the resident MVS NetView system and the remote NetView system. Both the resident MVS host and the remote NetView system have an operator ID, REMOPER, that you can customize with your own unique operator ID and profile name (*prfname*). This ID registers the NetView Bridge remote dispatcher as an management services (MS) application with the high performance transport API by executing the REMOTEBR command.

Note: It is not recommended that you define the NetView Bridge remote dispatcher on your resident MVS NetView system to be the same autotask as the NetView Bridge dispatcher. Your bridge dispatcher autotasks should be exclusive to either the NetView Bridge dispatcher or the NetView Bridge remote dispatcher.

To define the IDs, you must modify the DSIOPF member of DSIPARM. For example, in the following OPERATOR statement, replace the highlighted name (REMOPER) with the unique operator ID that you have selected, and replace *prfname* with the name of the profile that you have selected.

```
REMOPER OPERATOR PROFILEN=prfname
```

This change takes effect the next time you recycle the NetView program.

Step 2. Creating a NetView Bridge Remote Dispatcher Profile

You must create a new member in the NetView DSIPRF data set in MVS, a new file with *filetype* NCCFLST on the RUN disk for VM, or a new sublibrary member in PRD2.SAxx for VSE. This new member or file should have the same name (*prfname*) that you selected in Step 1 of your NetView Bridge remote access setup. It authorizes the operator to act as a NetView Bridge remote dispatcher autotask. This profile specifies the NetView Bridge command REMOTEBR as the initial command that runs at task initialization time to start NetView Bridge remote access.

Using the NetView-supplied sample DSIPROFF (CNMS1052), you can create a new profile by changing the existing PROFILE statement and naming it to the new profile name (*prfname*).

The syntax for this profile is:

prfname PROFILE IC=REMOTEBR

Notes:

1. For an explanation of the REMOTEBR command, see “Appendix A. NetView Bridge Initialization Commands” on page 87.
2. Replace *prfname* with the profile name that you defined in Step 1.
3. The autotask must have authority to issue the REMOTEBR command.

Step 3. Adding NetView Bridge Remote Access Command Model Statements to NetView

You must define the NetView Bridge remote access commands to the NetView program by using command model statements. These statements are in the DSICMD member of the DSIPARM data set in MVS, the CMS file DSICMD in VM, or the DSICMD member in the PRD2.SAxx sublibrary in VSE.

To access NetView Bridge remotely, you must define command model statements on both the resident NetView system and the remote NetView system.

- On the resident NetView system, the following CMDMDL statements are required:

```
RTRINIT    CMDMDL    MOD=DSINBINT,TYPE=R,RES=N
RTRQUEUE   CMDMDL    MOD=DSINBQUE,TYPE=R,RES=Y
TRANRCV    CMDMDL    MOD=DSINBRCV,TYPE=R,RES=Y
TRANSND    CMDMDL    MOD=DSINBSND,TYPE=R,RES=Y
DSINBRSM   CMDMDL    MOD=DSINBRSM,TYPE=R,RES=Y
DSINBTRM   CMDMDL    MOD=DSINBTRM,TYPE=R,RES=Y
DSINBR62   CMDMDL    MOD=DSINBR62,TYPE=R,PARSE=N,RES=Y
REMOTEBR   CMDMDL    MOD=DSINBREM,TYPE=R,PARSE=Y,RES=N
DSINBRLG   CMDMDL    MOD=DSINBRLG,TYPE=R,PARSE=Y,RES=Y
```

  Note: If the NetView Bridge dispatcher and remote dispatcher are infrequently used, consider changing RES=Y to RES=N to save storage. If only the remote dispatcher is infrequently used, consider changing RES=Y to RES=N on the DSINBR62 and DSINBRLG command model statements to save storage.

- The following CMDMDL statements are required on the remote NetView system:

```
TRANRCV    CMDMDL    MOD=DSINBRCV,TYPE=R,RES=Y
TRANSND    CMDMDL    MOD=DSINBSND,TYPE=R,RES=Y
DSINBR62   CMDMDL    MOD=DSINBR62,TYPE=R,PARSE=N,RES=Y
REMOTEBR   CMDMDL    MOD=DSINBREM,TYPE=R,PARSE=Y,RES=N
DSINBRLG   CMDMDL    MOD=DSINBRLG,TYPE=R,PARSE=Y,RES=Y
```

  Note: If the remote dispatcher is infrequently used, consider changing RES=Y to RES=N to save storage.

The REMOTEBR command is described in “Appendix A. NetView Bridge Initialization Commands” on page 87, and the TRANRCV and TRANSND commands are described in “Appendix B. NetView Bridge Requester API Commands and Service Routines” on page 91. Commands DSINBR62 and DSINBRLG are internal NetView commands and are not described.

Step 4. Creating NetView Command Procedures

For this step, you need to do some programming in the language of your choice. The NetView Bridge Requester API supports the following languages operating in MVS, VM, and VSE:

- C (MVS only)
- NetView command list language (VM and VSE only)
- PL/I (MVS only)
- REXX (MVS and VM only)

Using the criteria outlined in the *Tivoli NetView for OS/390 Customization Guide*, decide which programming language is most suitable for your needs. See "Chapter 4. Writing a NetView Bridge Command Procedure Using the NetView Bridge Requestor API" on page 55 for instructions on writing command procedures.

Chapter 3. Creating a Database Server

This chapter documents general-use programming interface and associated guidance information.

This chapter describes some considerations and procedures for developing your own database server. Creating your own database server requires an understanding of concepts that are unique to NetView Bridge. Some of these concepts are:

- Capturing return codes from the NetView Bridge server support API
- Compiling and link-editing a database server
- Creating database server control code using PL/I or C:
  - You should know how to request MVS system services from an assembler language program.
  - System programmer assistance is required to implement your database server in your MVS system because you need set up information, such as data set names.
- Creating transaction processors using PL/I or C:
  - You should have expertise in developing application programs using HLL programming languages (PL/I or C)
  - You should have an understanding of how to access your intended target database.

Documenting Your Database Server

Some procedures in this book direct you to documentation for your database server. If you have decided to create your own server, it is imperative that you document its flow and how it works, in detail.

Coding a Database Server with the Server Support API Using PL/I

This section provides the necessary information for coding a database server with the server support API using PL/I. Figure 13 on page 25 provides an example for you to use as a model when creating your own PL/I program.

Run-Time Options for PL/I

There are no restrictions on PL/I run-time options from the server support API.

Types of Parameters Passed to the Server Support API Service Routines for PL/I

Four types of parameters can be passed to the server support API service routines. They are:

- Pointer variables
- Integer variables
- Fixed-length character strings
- Varying-length character strings

The server support API service routines require you to declare, initialize, and pass parameters to the API service routines according to PL/I programming language specifications. For more information on these specifications, refer to*OS PL/I Version 2 Programming: Language Reference* and “Coding Your PL/I Program: Environment, Interfaces, and Restrictions” in *Tivoli NetView for OS/390 Customization: Using PL/I and C*.

Control Blocks for PL/I

A number of control blocks and include files are required for execution of your database server with the server support API service routines. You must include the NetView file DSIBPLI in your PL/I program. DSIBPLI includes the following files:

DSIBPHLB
Required file that contains a PL/I mapping of the NetView Bridge HLL control block.

DSIBPHLS
Required file that contains PL/I definitions for server support API service routines.

DSIBPCNM
Optional file that you can tailor to your needs. It declares return code constants for PL/I.

DSIPPRM
Required file only for server support service routine CNMESTR. It contains a PL/I mapping of the transaction parameter block.

Return Codes for PL/I

Upon completion of a server support API service routine, the return code from that service routine is stored in the return code field (HLBRC) of the control block DSIBPHLB. Your code must check this field after each API service routine invocation.

Coding a Database Server with the Server Support API Using C

This section provides the necessary information for coding a database server with the server support API using C programming language. Figure 14 on page 42 provides an example for you to use as a model when creating your own C program.

Run-Time Options for C

There are no restrictions on C run-time options from the server support API.

Types of Parameters Passed to the Server Support API Service Routines for C

Four types of parameters can be passed to the server support API service routines. They are:

- Pointer variables
- Integer variables
- Fixed-length character strings
- Varying-length character strings

Declare, initialize, and pass parameters to the API service routines according to C programming language specifications. For more information on these specifications, refer to *SAA®: Common Programming Interface C Reference* and “Coding Your PL/I Program: Environment, Interfaces, and Restrictions” in *Tivoli NetView for OS/390 Customization: Using PL/I and C*.

Control Blocks for C

A number of control blocks and include files are required for execution of your C program with the server support API service routines. You must include the NetView file DSIBC in your C program. DSIBC includes the following files:

DSIBCHLB
Required file that contains a C mapping of the NetView Bridge HLL control block.

DSIBCCAL
Required file that contains C definitions for server support API service routines.

DSIBCCNM
Optional file that you can tailor to your needs. It declares return code constants for C.

DSICPRM
Required file only for server support service routine CNMESTR. It contains a C mapping of the transaction parameter block.

DSICVARC
Optional file that contains a C mapping of a varying-length character string.

Return Codes for C

Upon completion of a server support API service routine, the return code from that service routine is stored in the return code field (HLBRC) of the control block DSIBCHLB. Your code must check this field after each API service routine invocation.

Creating Database Server Control Code

This section describes the procedures for creating your own database server control code. The database server control code must initialize the database server and repeat the following steps as necessary:

1. Wait for a NetView Bridge transaction request
2. Invoke the appropriate transaction processor when a transaction request is received.

This code must also terminate the database server, when appropriate.

Initializing the Database Server

Your code must initialize the database server by performing the following procedures:

- Open any necessary files. For example, you might need to open or access logging files, initialization parameter files, or files related to your target database.
- Establish a mechanism for recognizing when an MVS operator wants to stop your database server. One method for doing this is to code your server to recognize the MVS STOP command. This provides for an orderly termination process for your database server with NetView Bridge.
- Initialize the NetView Bridge server support API by calling the CNMETIN service routine.

  CNMETIN performs the following functions:

  – Defines the environment by establishing the *bhlbptr* parameter used by all of the other server support API service routines.
  – Initializes the server support API by defining your program-to-program interface (PPI) input queue. The name for the input queue is defined by the *tiinq* parameter of CNMETIN.

Notes:

1. If you intend to use multiple servers, the name that you assign to *tiinq* must be unique for each copy of a database server.
2. The limit for queued transactions is defined by the *tiinqlim* operand of CNMETIN. You must assign queue limits for your database servers. Select an adequate queue size to avoid a problem with the queue filling up during database server operations. This number should be approximately four times the number of concurrent ready tokens that can be pending for this database server. The number of concurrent ready tokens that can be pending for this database server depends on the number specified on the *rqueue* parameter of RTRINIT.

– Identifies the program-to-program interface (PPI) queue for communication to the NetView program. The name for this output queue is defined by the *tioutq* operand of CNMETIN. This name must be identical to the name specified with the *oqueue* operand of the NetView Bridge command RTRINIT.

Note: All nonzero CNMETIN return codes should be treated as catastrophic failures. In particular, return code (RET_PPI_FAILURE +16) is generated if you attempt to initialize a database server that is already started.

- You might have to initialize your target database interface. You must be familiar with the interface requirements of your target database. Consult any available documentation for your database and create the necessary code for initializing it, if required.

Controlling Flow

The control code must control the flow of data between your database server and the NetView program. Establishing control of this interface involves several considerations. These are described in the following sections.

Sending Ready Tokens

Some database interfaces can support multitasking. If this is the case at your installation, your database server can send multiple ready tokens. Consult any available documentation for your database interface for information on its limitations on multitasking.

You might consider receiving and processing any transaction requests in the PPI queue *tiinq*. Before your code issues any ready tokens, these transaction requests should be passed to the appropriate transaction processors.

Your code must call the NetView Bridge server support API CNMERTR service routine once for each transaction that can be processed concurrently. (If your database interface does not allow concurrent transactions, CNMERTR needs to be called only once.) This routine sends a ready token to NetView Bridge.

When it receives a ready token indicating that this copy is ready to process a transaction, NetView Bridge sends a transaction to this copy of your database server.

Receiving a Queued Transaction

To receive the next transaction in the input queue, you must call the CNMENTR service routine. CNMENTR attempts to retrieve the next transaction. If no transaction is found in the input queue, return code RET_NOT_FOUND is issued, and you need to enter a wait state.

Waiting for a Transaction

To enter a wait state after an unsuccessful attempt to get the next transaction, your code must call the NetView Bridge CNMEWAT service routine. In addition to waiting for a transaction to be placed on the input queue, CNMEWAT can also wait for events specific to your server (for example, the MVS operator command STOP).

Your code can set the WAECBP parameter of service routine CNMEWAT to the address of a standard event control block (ECB) list to wait for user-defined events. If you do not want to wait for any user-defined events, set parameter WAECBP to a PL/I or C null. For an example of an assembler language source code that you can use to obtain the address of a communications ECB, see Figure 15 on page 52.

Note: Parameter WAECBP must be set to either an ECB address or a PL/I or C null. When an awaited event is posted, CNMEWAT issues one of two return codes:

- RET_WAIT_POSTED (RC=12)

  This return code means that one or more of the user-provided ECBs were posted, and that there is no transaction in the input queue. You are responsible for handling the processing of your ECBs.
- RET_WAIT_POSTED_PPI (RC=8)

  This return code means that at least one transaction is in the input queue. One or more of the user-provided ECBs might be posted.

If you have established a mechanism to recognize that the MVS operator wants to stop the server, you should call the CNMETQU service routine to send a CNMQUISE transaction to the NetView Bridge dispatcher. This removes all ready tokens related to this database server. When the dispatcher completes this task, it sends the CNMQUISE transaction back to the database server to indicate that the quiesce state is in effect.

Obtaining a Transaction Identifier

Once CNMENTR successfully retrieves a transaction, your code can call the CNMEGTP service routine to determine what transaction was received.

CNMEGTP can retrieve many parameters. At this point, however, you need to use it only to identify the transaction that was received. For this identification to take place, call CNMEGTP with its parameters set as follows:

- The *gpmethod* parameter must have a value of H (meaning this is a header parameter).
- The *gpnameh* parameter must have a value of TRANSID (meaning that you want it to return the transaction identifier).
- The *gpdatlen* parameter must contain the size of the data buffer that receives the name of this transaction. The value of this parameter must be at least 8.
- The *gpdata* parameter must be a varying-length character field that is used to receive the incoming transaction ID.

Invoking the Appropriate Transaction Processor

Using the transaction identifier that you received, invoke the appropriate transaction processor for your database server.

After the transaction processor completes its work, your code must call the CNMERTR service routine to inform the NetView Bridge dispatcher that you are ready to process another transaction. Your code must then return to receive the next transaction (see “Receiving a Queued Transaction” on page 18).

If the transaction identifier is CNMQUISE, the NetView Bridge dispatcher removes all ready tokens, and the database server does not receive any more transactions. After the last transaction completes processing, you should invoke the termination process described in “Terminating the Database Server”.

Terminating the Database Server

Because all transaction processing is complete, your code must perform the following:

- Call the NetView Bridge server support API CNMETRM service routine to terminate the NetView Bridge server support API.
- Terminate your database interface, as required by your database.
- Close all data sets.
- Issue a message to the MVS system console indicating that this server is terminating.

Creating a Transaction Processor

This section describes how to create a transaction processor to work between the NetView Bridge server support API and your target database.

After you design the transaction, you can develop the transaction processor. Your transaction processor must be written to accept a transaction request from the server support API. The processor must then extract the input parameters from the transaction request and use them to interact with the interface to your target database.

Once the transaction request is satisfied by the target database, your transaction processor must process the results and convert them to the format required by the server support API. Your transaction processor might then pass the transaction reply to the server support API, if a reply was requested.

Designing the Transaction

Before you create a transaction processor, you must make some basic design decisions. These decisions include:

- Defining the function of the transaction
- Deciding what input parameters are required by the transaction processor and the target database
- Deciding what needs to be returned in the transaction reply

Ensure that you are familiar with language restrictions pertaining to the NetView Bridge command procedures that are requesting the services of your transaction processor. In particular, NetView command list language parameters cannot exceed 11 characters, and this language has no provisions for array processing. For more information on the NetView command list language, refer to *Tivoli NetView for OS/390 Customization: Using REXX and the NetView Command List Language*.

Processing the Transaction Request

To extract the input parameters from the transaction request, you must do two things:

1. Write the code to extract header parameters (method H) using the NetView Bridge server support API CNMEGTP service routine. (Methods for retrieving parameters are described in “CNMEGTP (CNMETRPARM) Service

Routine—Get Transaction Request Parameters" on page 107.) Extract the following header parameters for use in generating the reply:

- TRANSID (transaction ID)
- ORIGNET (originating network ID)
- ORIGDOM (originating domain ID)
- ORIGTASK (originating task ID)
- DESTNET (destination network ID)
- DESTDOM (destination domain ID)
- DESTTASK (destination task ID)
- TRTYPE (type of transaction)
- CORR (correlator)
- VOCAB (vocabulary)

2. Write the code to extract the transaction parameter (method N or P) information using the NetView Bridge server support API CNMEGTP service routine. See "CNMEGTP (CNMETRPARM) Service Routine—Get Transaction Request Parameters" on page 107 for details on how to use this service routine.

Interacting with the Database Interface

Write code that sends the transaction request to the target database interface. You must ensure that the data you forward meets any special requirements of your target database. Consult the documentation for your target database for any such requirements.

Once the transaction request is forwarded to the database interface and the request is satisfied, your transaction processor must process the results in preparation for conversion and forwarding to the NetView Bridge server support API.

Setting Up the Transaction Parameter Blocks

The NetView program contains a parameter block structure specifically for use with the NetView Bridge server support API CNMESTR (send a transaction reply) service routine. The parameter block structure has two separate versions—one for PL/I (DSIPPRM), and one for C (DSICPRM).

When you write the code to initialize your parameters, place each nonheader parameter in its own allocated copy of the parameter block structure. Your code must store the required information in this parameter block structure.

Finally, write code to link these parameter block structures together for use by the CNMESTR service routine. To do this, set the pointer field (PRMLINK) in each parameter block to the next parameter block. The final parameter block should point to null. This links the parameter blocks together, as shown in Figure 8.

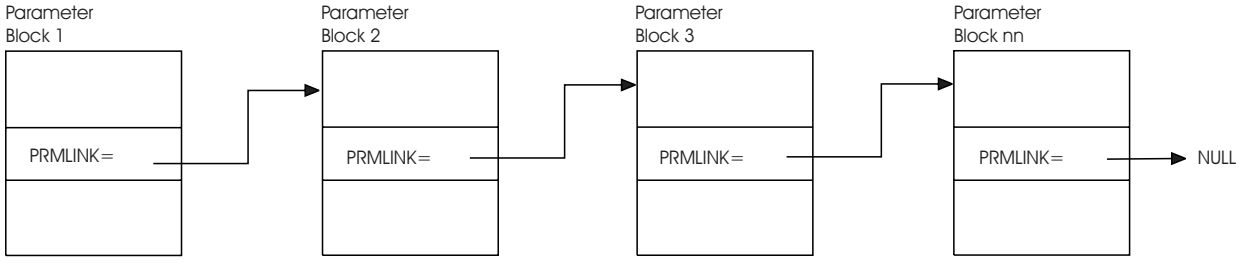


Figure 8. Linking Parameter Blocks to Form a Parameter Block Structure

Forwarding the Transaction Reply

Now, create the code to forward the data (the transaction reply) back to the NetView program using the server support API. This code must perform the following tasks:

- Examine the value of the *trtype* parameter that was extracted from the original transaction request. The *trtype* must contain one of the following three values:
  - QNON—No reply is to be sent.
  - QERR—Reply is sent only if errors have occurred during processing.
  - QALL—All data is to be returned.
- Call the NetView Bridge server support API CNMESTR service routine to send the transaction reply to the NetView requestor if a reply is to be sent (*trtype* of QERR or QALL).

Your call to CNMESTR must contain several header operands. Table 4 lists these operands and their sources.

Table 4. Header Operands Required for the CNMESTR Service Routine

CNMESTR Operands	Source
transid	*transid* operand of the transaction request
trtype	RPLY
destnet	*orignet* operand of the transaction request
destdom	*origdom* operand of the transaction request
desttask	*origtask* operand of the transaction request
orignet	*destnet* operand of the transaction request
origdom	*destdom* operand of the transaction request
origtask	*desttask* operand of the transaction request
vocab	Can be the *vocab* operand of the transaction request (only used if your database server supports this function)
corr	*corr* operand of the transaction request
respcode	Code generated by the transaction processor that summarizes the results of the processed transaction
parms	Pointer to the first transaction parameter block in the linked list

Once you have created the code to perform the tasks outlined in this section, your transaction processor is complete.

Compiling and Link-Editing a Database Server with the Server Support API in PL/I

This section describes the procedures for compiling and link-editing your database server written in PL/I with the NetView Bridge server support API.

Compiling Considerations for PL/I

The recommended compile-time options for the NetView Bridge server support API are MACRO, LIST, and OBJECT. You must include SYS1.MACLIB in the SYSLIB statement of the compiled job control language (JCL). An example of the compile-step JCL for PL/I is shown in Figure 9.

```
//COMPILE  EXEC  PGM=IEL0AA,REGION=1000K,
//               PARM='OBJECT,MACRO,LIST'
                 •
                 •
                 •
//SYSLIB   DD    DSN=SYS1.MACLIB,DISP=SHR
                 •
                 •
                 •
```

Figure 9. Example of a Compile Step JCL for the Server Support API in PL/I

Link-Editing Considerations for PL/I

The following rules apply to link-editing PL/I database server modules with the NetView Bridge server support API modules:

- PL/I load modules can reside in either 24- or 31-bit storage and can be entered in either addressing mode.
- You must link-edit your PL/I load modules with the CNMETIN and CNMETRM load modules provided by the NetView program in SYS1.LINKLIB.
- Include SYS1.LINKLIB in the SYSLIB statement in the PL/I link-edit step of the JCL.

An example of the link-edit step of the JCL for PL/I is shown in Figure 10.

```
//LKED  EXEC  PGM=IEWL,
//            PARM='XREF,RENT,LET,LIST',
//            REGION=4096K,COND=(8,LE,COMPILE)
              •
              •
              •
//SYSLIB DD   DSN=SYS1.PLIBASE,DISP=SHR
//       DD   DSN=SYS1.SIBMBASE,DISP=SHR
//       DD   DSN=SYS1.LINKLIB,DISP=SHR
              •
              •
              •
  INCLUDE     SYSLIB(CNMETIN)
              •
              •
  INCLUDE     SYSLIB(CNMETRM)
  MODE        AMODE(31),RMODE(ANY)
  NAME        DBSERVER(R)
```

Figure 10. Example of a Link-Edit Step JCL for the Server Support API in PL/I

Compiling and Link-Editing a Database Server with the Server Support API in C

This section describes the procedures for compiling and link-editing your database server written in C with the NetView Bridge server support API.

Compiling Considerations for C

Include SYS1.MACLIB in the SYSLIB statement of the compiled JCL. An example of the compile-step JCL for C is shown in Figure 11.

```
//COMPILE  EXEC  PGM=EDCCOMP,REGION=&REGSIZ
                 •
                 •
                 •
//SYSLIB   DD    DSN=SYS1.MACLIB,DISP=SHR
                 •
                 •
                 •
```

Figure 11. Example of a Compile Step JCL for the Server Support API in C

Link-Editing Considerations for C

The following rules apply for link-editing C database server modules with the NetView Bridge server support API modules:

- C load modules can reside in either 24- or 31-bit storage and can be entered in either addressing mode.
- Link-edit your C load modules with the CNMETIN and CNMETRM load modules provided by the NetView program in SYS1.LINKLIB.
- Include SYS1.LINKLIB in the SYSLIB statement in the C link-edit step of the JCL.

An example of the link-edit step of the JCL for C is shown in Figure 12.

```
//LKED  EXEC  PGM=IEWL,
//            PARM='XREF,RENT,LET,LIST',
//            REGION=4096K,COND=(8,LE,COMPILE)
                •
                •
                •
//SYSLIB DD   DSN=SYS1.V1R1M1.SEDCBASE,DISP=SHR
//       DD   DSN=SYS1.V2R2M0.SIBMBASE,DISP=SHR
//       DD   DSN=SYS1.LINKLIB,DISP=SHR
                •
                •
                •
  INCLUDE     SYSLIB(CNMETIN)
                •
                •
  INCLUDE     SYSLIB(CNMETRM)
  ENTRY       CEESTART
  MODE        AMODE(31),RMODE(ANY)
  NAME        DBSERVER(R)
```

Figure 12. Example of a Link-Edit Step JCL for the Server Support API in C

Example PL/I Program Using the Server Support API

Figure 13 on page 25 is an example of a PL/I program that uses the server support API. You can use this as a model for creating your own PL/I program.

```
SERTRAN: PROC OPTIONS(MAIN);
/*********************************************************************/
/* Sample PL/I program using the Server Support API.                 */
/*********************************************************************/

/*********************************************************************/
/* Include files section                                             */
/* DSIBPLI will include the following files:                         */
/*    DSIBPHLB - PL/I mapping of HLB                                 */
/*    DSIBPHLS - PL/I definition for Server Support API routines     */
/*    DSIBPCNM - Return codes                                        */
/*    DSIPPRM  - PL/I mapping of transaction parameter block         */
/*********************************************************************/
%INCLUDE DSIBPLI;

/*********************************************************************/
/* External module declaration                                       */
/*********************************************************************/
DCL GETECB ENTRY (PTR,CHAR(8));
/*********************************************************************/
/* Constants                                                         */
/*********************************************************************/
DCL EBCDIC_TYPE   BIT(16)         INIT('0000000011101110'B);
DCL BIN_TYPE      BIT(16)         INIT('0000000010111011'B);
DCL NOT_AN_INDEX  FIXED BIN(31,0) INIT(-32767);

/*********************************************************************/
/* Variables used to call CNMETIN service routine                    */
/*********************************************************************/
DCL TIINQ_NAME    CHAR(8)         INIT('SERINQ');
                                          /* Server input queue  */
DCL TIOUTQ_NAME   CHAR(8)         INIT('OQUEUE');
                                          /* Server output queue */
DCL TIINQ_LIMIT   FIXED BIN(31,0) INIT(4);

/*********************************************************************/
/* Variables used to call CNMEGTP service routine                    */
/*********************************************************************/
DCL GPMETHOD      CHAR(1);
DCL GPNAMEH       CHAR(8);
DCL GPNAMEP       CHAR(31) VARYING;
DCL GPINDEX       FIXED BIN(31,0) INIT(NOT_AN_INDEX);
DCL GPDATA        CHAR(256) VARYING;
DCL GPDATLEN      FIXED BIN(31,0);
DCL GPTYPE        CHAR(2)         INIT(EBCDIC_TYPE);
DCL GPCURSOR      FIXED BIN(31,0);
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 1 of 17)

```
/*********************************************************************/
/* Variables used to call CNMESTR service routine                    */
/*********************************************************************/
DCL TRANSID       CHAR(8);
DCL TRTYPE        CHAR(4)         INIT('RPLY');
DCL DESTNET       CHAR(8);
DCL DESTDOM       CHAR(8);
DCL DESTTASK      CHAR(8);
DCL ORIGNET       CHAR(8);
DCL ORIGDOM       CHAR(8);
DCL ORIGTASK      CHAR(8);
DCL VOCAB         CHAR(8);
DCL CORR          CHAR(8);
DCL RESPCODE      CHAR(8);
DCL PARMS         PTR            INIT(NULL());

/*********************************************************************/
/* Variables used to process ECB list                                */
/*********************************************************************/
DCL CECB_PTR      PTR;
DCL CECB_BIT_FORM(32) BIT BASED (CECB_PTR);
DCL WAECBP        PTR;
DCL JOB_NAME      CHAR(8);
DCL ECB           FIXED BIN(31) BASED;
DCL USRECB_POSTED BIT(1)          INIT('0'B);

/*********************************************************************/
/* Other variables used in the program as global variables           */
/*********************************************************************/
DCL TERMINATED    BIT(1) INIT('0'B);
                                  /* '0'B Initially               */
                                  /* '1'B When program receives   */
                                  /*      a QUIESCE token         */
DCL EMPTYQ        BIT(1) INIT('0'B);
                                  /* '0'B When initialized to     */
                                  /*      NetView Bridge Server   */
                                  /*      Support API.  There may */
                                  /*      be outstanding transact-*/
                                  /*      ions in the input queue.*/
                                  /* '1'B When finished reading   */
                                  /*      all outstanding messages*/
                                  /*      in the output queue and */
                                  /*      is ready for work.      */
DCL TRANS_TYPE    CHAR(4) INIT (' ');
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 2 of 17)

```
/*********************************************************************/
/* Initialization of your database interface.  NOT shown here       */
/*********************************************************************/

/*********************************************************************/
/* Initialize the NetView Bridge Server Support API                  */
/*********************************************************************/
CALL CNMETIN (BHLBPTR,             /* To define the environment     */
              TIINQ_NAME,          /* Input queue name              */
              TIINQ_LIMIT,         /* Limit queued transactions     */
              TIOUTQ_NAME);        /* Output queue name             */

IF (HLBRC = RET_GOOD) THEN         /* Initialization is successful  */
  DO;
    /*****************************************************************/
    /* Clean up the input queue by reading and discarding all        */
    /* outstanding transactions in the input queue.  Processing of   */
    /* outstanding transactions and error handling are NOT shown     */
    /* here.                                                         */
    /*****************************************************************/
    DO WHILE ((HLBRC = RET_GOOD) & (¬EMPTYQ));
      CALL CNMENTR (BHLBPTR);
      IF (HLBRC = RET_NOT_FOUND) THEN
        EMPTYQ = '1'B;
    END;


    /*****************************************************************/
    /* The input queue is now empty.  Prepare the Event Control      */
    /* Block (ECB) list to be used later when calling CNMEWAT.       */
    /*   1. Call GETECB to get the pointer to the communication      */
    /*      ECB and the job name of the database server.             */
    /*      (See the assembler example GETECB in this document.)     */
    /*   2. Turn the first bit of the pointer to the communication   */
    /*      ECB list to 1 to indicate the end of the CECB list.      */
    /*   3. Get the address of the pointer to CECB for use later by  */
    /*      CNMEWAT.                                                 */
    /*****************************************************************/
    IF EMPTYQ THEN
      DO;                          /* If input queue is empty       */
        CALL GETECB(CECB_PTR,JOB_NAME);
        CECB_BIT_FORM(1) = '1'B;
        WAECBP = ADDR(CECB_PTR);
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 3 of 17)

```
/*************************************************************/
/* Do the following process until terminated                 */
/*   1. Send a READY token                                   */
/*   2. Wait for ECB list to be posted                       */
/*   3. Wake up when ECB list is posted to either:           */
/*     (i) process a transaction    OR                       */
/*     (ii) send the QUIESCE token when ECB is terminated    */
/*************************************************************/
CALL CNMERTR (BHLBPTR);    /* Send a READY token             */
IF (HLBRC = RET_GOOD) THEN
  DO UNTIL (TERMINATED);
    CALL CNMEWAT (BHLBPTR,WAECBP); /* Wait for the ECB list */

    /*******************************************************/
    /* Wake up to check which ECB is posted                */
    /*******************************************************/
    SELECT (HLBRC);
      WHEN (RET_WAIT_POSTED_PPI)
        DO;
          /***********************************************/
          /* Either PPI ECB or User ECB (STOP command is    */
          /* entered) is posted.  If User ECB is posted,    */
          /* send a QUIESCE token and process all transact- */
          /* ions in the input queue.  Only send READY      */
          /* token when it is not User ECB posted.          */
          /***********************************************/
          IF (SUBSTR(UNSPEC(CECB_PTR -> ECB),2,1) = '1'B) THEN
                        /* User ECB is posted   */
            DO;
              USRECB_POSTED = '1'B;
              CALL CNMETQU (BHLBPTR);
            END;

          CALL PROCESS_TRANSACTION;
          IF (¬ USRECB_POSTED) THEN
            /*********************************************/
            /* Send a READY token ONLY if the user ECB is   */
            /* NOT posted.                                  */
            /*********************************************/
            CALL CNMERTR (BHLBPTR);
        END;
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 4 of 17)

```
            WHEN (RET_WAIT_POSTED)
              DO;
                /************************************************/
                /* Operator has entered the STOP command which  */
                /* caused this ECB to be posted.  Send a QUIESCE */
                /* token.                                       */
                /************************************************/
                CALL CNMETQU (BHLBPTR);
              END;

            OTHERWISE;
           END;                    /* SELECT on HLBRC             */
         END;                      /* Do until TERMINATED         */
      END;                         /* If input queue is empty     */
  END;                             /* Initialization is successful */
/*********************************************************************/
/* Initialization of NetView Bridge Server is not successful         */
/*********************************************************************/
ELSE
  DO;
                                   /* Error processing NOT shown   */
  END;

/*********************************************************************/
/* PROCESS_TRANSACTION procedure                                     */
/*   Process all transactions in the input queue.  This procedure    */
/*   is intended only as an example of how to process transactions   */
/*   in the input queue.  Four other procedures will be called from  */
/*   here:                                                           */
/*     1. GET_HEADER: to illustrate how to obtain all header         */
/*                    information                                    */
/*     2. GET_NEXT:   to illustrate how to use CNMEGTP to get the    */
/*                    next parameter in the parameter list           */
/*     3. GET_NAME:   to illustrate how to use CNMEGTP to get the    */
/*                    value of a parameter by name                   */
/*     4. SEND_REPLY: to illustrate how to use CNMESTR to send a     */
/*                    reply to the NetView requester                 */
/*********************************************************************/
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 5 of 17)

```
PROCESS_TRANSACTION: PROC;

 DO WHILE (HLBRC = RET_GOOD);
   CALL CNMENTR(BHLBPTR);          /* Get a transaction            */

   SELECT (HLBRC);
     WHEN (RET_GOOD)
       DO;
         CALL GET_HEADER;
         IF ((HLBRC = RET_GOOD) & (¬TERMINATED)) THEN
           DO;
             CALL GET_NEXT;
             IF (HLBRC = RET_GOOD) THEN
               DO;
                 CALL GET_NAME;
                 IF (HLBRC = RET_GOOD) THEN
                   CALL SEND_REPLY;
               END;
           END;
       END;

     OTHERWISE;
   END;                            /* SELECT on HLBRC              */
 END;                              /* Do while                     */
END PROCESS_TRANSACTION;

/*********************************************************************/
/* GET_HEADER procedure                                              */
/*   This procedure is intended only as an example of how to use     */
/*   the CNMEGTP service routine with GPMETHOD = 'H' (to get the     */
/*   header parameters of a transaction.                             */
/*                                                                   */
/*   Header parameters are:                                          */
/*     - ORIGNET - Origin network identifier                         */
/*     - ORIGDOM - Origin domain name                                */
/*     - ORIGTASK- Origin task name                                  */
/*     - DESTNET - Destination network identifier                    */
/*     - DESTDOM - Destination domain name                           */
/*     - DESTTASK- Destination task name                             */
/*     - TRANSID - Transaction identifier                            */
/*     - TRTYPE  - Transaction type                                  */
/*     - VOCAB   - Vocabulary, user specific                         */
/*     - CORR    - Correlator, user specific                         */
/*                                                                   */
/*   The following parameters of CNMEGTP: GPNAMEP, GPINDEX and       */
/*   GPCURSOR are NOT used when getting header parameters.  As a     */
/*   result, they will NOT be initialized.                           */
/*********************************************************************/
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 6 of 17)

```
GET_HEADER: PROC;

  /*****************************************************************/
  /* Local variables                                               */
  /*****************************************************************/
  DCL HDR_COUNT        BIN(31);    /* Header counter               */
  DCL ORIGIN_TASK_NAME CHAR(8);
  DCL TRANSID_NAME     CHAR(8);

  /*****************************************************************/
  /* Procedure starts here                                         */
  /*****************************************************************/
  GPMETHOD = 'H';
  HDR_COUNT= 1;
  DO WHILE ((HLBRC = RET_GOOD) &
            (HDR_COUNT <= 10)  &
            (¬TERMINATED));
    SELECT (HDR_COUNT);
      WHEN (1)   GPNAMEH = 'ORIGNET';
      WHEN (2)   GPNAMEH = 'ORIGTASK';
      WHEN (3)   GPNAMEH = 'ORIGDOM';
      WHEN (4)   GPNAMEH = 'DESTNET';
      WHEN (5)   GPNAMEH = 'DESTTASK';
      WHEN (6)   GPNAMEH = 'DESTDOM';
      WHEN (7)   GPNAMEH = 'TRANSID';
      WHEN (8)   GPNAMEH = 'VOCAB';
      WHEN (9)   GPNAMEH = 'CORR';
      WHEN (10)  GPNAMEH = 'TRTYPE';
      OTHERWISE;
    END;

    /*****************************************************************/
    /* Call CNMEGTP to get the header                                */
    /*****************************************************************/
    CALL CNMEGTP(BHLBPTR,            /* to define the environment   */
                 GPMETHOD,           /* retrieve header parameter   */
                 GPNAMEH,            /* header name                 */
                 GPNAMEP,            /* not use in this case        */
                 GPINDEX,            /* not use in this case        */
                 GPDATA,             /* output header value         */
                 GPDATLEN,           /* output header length        */
                 GPTYPE,             /* not used in this case       */
                 GPCURSOR);          /* not used in this case       */
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 7 of 17)

```
IF (HLBRC = RET_GOOD) THEN
  SELECT (HDR_COUNT);
    WHEN (2)
      DO;
      /*********************************************************/
      /* Save the origin task name for later use (to determine  */
      /* if a quiesce token was received)                       */
      /*********************************************************/
        ORIGIN_TASK_NAME = SUBSTR(GPDATA,1,BHLBPTR -> HLBLENG);
       END;

    WHEN (7)
      DO;
        TRANSID_NAME = SUBSTR(GPDATA,1,BHLBPTR -> HLBLENG);
        SELECT (TRANSID_NAME);
          WHEN ('CREATE')
            DO;
              /************************************************/
              /* Call the appropriate routine to create a       */
              /* record in the target database                  */
              /************************************************/
            END;

          WHEN ('SEARCH')
            DO;
              /************************************************/
              /* Call the appropriate routine to search for a   */
              /* particular record in the target database       */
              /************************************************/
            END;

          WHEN ('UPDATE')
            DO;
              /************************************************/
              /* Call the appropriate routine to update a       */
              /* record in the target database                  */
              /************************************************/
            END;
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 8 of 17)

```
            /*******************************************************/
            /* A QUIESCE token has been received                   */
            /*******************************************************/
            WHEN ('CNMQUISE')
              DO;
                IF (ORIGIN_TASK_NAME = TIINQ_NAME) THEN
                  DO;
                    TERMINATED = '1'B;
                    CALL CNMETRM (BHLBPTR);
                  END;
              END;

             OTHERWISE;
           END;                    /* Select on TRANSID_NAME        */
         END;

        WHEN (10)
          DO;
            /*******************************************************/
            /* Save the transaction type to be used later.         */
            /*******************************************************/
            TRANS_TYPE = SUBSTR(GPDATA,1,BHLBPTR -> HLBLENG);
          END;
        OTHERWISE;
      END;                         /* Select on HDR_COUNT           */

    /*****************************************************************/
    /* Increment HDR_COUNT to continue the loop                      */
    /*****************************************************************/
    HDR_COUNT = HDR_COUNT + 1;
  END;
END GET_HEADER;
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 9 of 17)

```
/*********************************************************************/
/* GET_NEXT procedure                                                */
/*   This procedure is intended only as an example of how to use     */
/*   the CNMEGTP service routine with GPMETHOD = 'N' (to retrieve    */
/*   the name and value of the next transaction parameter) starting  */
/*   the search process from GPCURSOR.                               */
/*                                                                   */
/*   Parameters for call to CNMEGTP:                                 */
/*     GPMETHOD = 'N' Input parameter                                */
/*     GPCURSOR = 0   indicates that the search process is to begin  */
/*                    at the start point of the parameter list.      */
/*                    Otherwise, the search process is determined    */
/*                    by GPCURSOR.  This is an input parameter to    */
/*                    CNMEGTP.  Users shall NOT modify this          */
/*                    parameter.                                     */
/*     GPNAMEP        Output parameter to contain the name of the    */
/*                    transaction parameter.                         */
/*     GPINDEX        Output parameter to indicate whether the       */
/*                    parameter is an element of an array.  It's     */
/*                    value will be -32767 if NOT an element of an   */
/*                    array.                                         */
/*     GPDATA         Output parameter to contain the value of the   */
/*                    transaction parameter being retrieved.         */
/*     GPDATLEN       Input parameter.  It specifies the length of   */
/*                    GPDATA (in bytes).  The actual length of the   */
/*                    data returned in GPDATA will be stored in      */
/*                    HLBLENG.                                       */
/*     GPTYPE         Output parameter that specifies the type of    */
/*                    the transaction parameter.  Its value will be  */
/*                    either '00EE'X or '00BB'X.                     */
/*                                                                   */
/*   The following parameter of CNMEGTP: GPNAMEH is not used when    */
/*   retrieving the next parameter.  As a result it will NOT be      */
/*   initialized here.                                               */
/*********************************************************************/
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 10 of 17)

```
GET_NEXT: PROC;

  /*****************************************************************/
  /* Initialize input parameters to CNMEGTP.                       */
  /*****************************************************************/
  GPCURSOR = 0;                 /* Indicate that search is to start*/
                                /*   at the beginning of the list  */
  GPMETHOD = 'N';
  GPDATLEN = 256;                 /* Since GPDATA is defined as    */
                                  /* CHAR(256) VARYING             */

  DO WHILE ((HLBRC = RET_GOOD) |
            (HLBRC = RET_DATA_TRUNC));/* output value is truncated  */
    CALL CNMEGTP (BHLBPTR,
                  GPMETHOD,
                  GPNAMEH,
                  GPNAMEP,
                  GPINDEX,
                  GPDATA,
                  GPDATLEN,
                  GPTYPE,
                  GPCURSOR);
    /*****************************************************************/
    /* Process the output -- This is only a skeleton.  The           */
    /* following procedures will be specific to your database        */
    /* server.  Those return codes that are NOT mentioned below,     */
    /* are unlikely to occur.                                        */
    /*****************************************************************/
    SELECT (HLBRC);
      WHEN (RET_GOOD)
        DO;
          /*************************************************************/
          /* Successful retrieval of transaction parameter             */
          /*************************************************************/
        END;

      WHEN (RET_END_FILE)
        DO;
          /*************************************************************/
          /* No more parameters to be retrieved                        */
          /*************************************************************/
        END;

      WHEN (RET_BAD_LENGTH)
        DO;
          /*************************************************************/
          /* Re-check GPDATLEN                                         */
          /*************************************************************/
        END;
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 11 of 17)

```
      WHEN (RET_BAD_CURSOR)
        DO;
          /*********************************************************/
          /* Re-check GPCURSOR.  GPCURSOR should never be altered  */
          /* by the user.                                          */
          /*********************************************************/
        END;

      WHEN (RET_BAD_METHOD)
        DO;
          /*********************************************************/
          /* Re-check GPMETHOD.  Valid values are 'H', 'P' and 'N'. */
          /*********************************************************/
        END;

      WHEN (RET_DATA_TRUNC)
        DO;
          /*********************************************************/
          /* GPDATA is NOT large enough to contain the value of the */
          /* transaction parameter.                                */
          /*                                                       */
          /* GPCURSOR is updated to point to the next parameter.   */
          /* If the user wishes to retrieve the current truncated  */
          /* parameter value again, he/she is recommended to follow */
          /* these steps:                                          */
          /*                                                       */
          /*   1. Save the name of the parameter and its index,    */
          /*      GPNAMEP and GPINDEX, respectively.  Do NOT alter */
          /*      GPCURSOR.  It will be used later.                */
          /*   2. Allocate more space for GPDATA.  The additional  */
          /*      amount of space is equal to the difference       */
          /*      between HLBLENG and GPDATLEN.                    */
          /*   3. Update GPDATLEN to reflect GPDATA's new size.    */
          /*   4. Call CNMEGTP with GPMETHOD of 'P' with the saved */
          /*      values for GPNAMEP and GPINDEX (Step 1).         */
          /*   5. Check the output as usual.                       */
          /*   6. Continue to call CNMEGTP with GPMETHOD of 'N' and */
          /*      the unaltered GPCURSOR (from the previous call to */
          /*      get next parameter) until there are no more      */
          /*      parameters to receive.                           */
          /*********************************************************/
        END;
      END;                        /* Select on HLBRC               */
    END;                          /* Do while HLBRC = RET_GOOD ...  */
END GET_NEXT;
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 12 of 17)

```
/*********************************************************************/
/* GET_NAME procedure                                                */
/*   This procedure is intended only as an example of how to use     */
/*   the CNMEGTP service routine with GPMETHOD = 'P' (to retrieve    */
/*   the value of a transaction parameter) when its name is known.   */
/*                                                                   */
/*   Parameters for call to CNMEGTP:                                 */
/*     GPMETHOD = 'P' Input parameter.                               */
/*     GPNAMEP        Input parameter.  It contains the name of the  */
/*                    parameter whose value is retrieved.            */
/*     GPINDEX        Input parameter.  It indicates whether or not  */
/*                    the name specified by GPNAMEP is an element    */
/*                    of an array.  Its value will be -32767 if NOT  */
/*                    an element of an array.                        */
/*     GPDATA         Output parameter to contain the value of the   */
/*                    transaction parameter being retrieved.         */
/*     GPDATLEN       Input parameter.  It specifies the length of   */
/*                    GPDATA (in bytes).  The actual length of the   */
/*                    data returned in GPDATA will be stored in      */
/*                    HLBLENG.                                       */
/*     GPTYPE         Output parameter that specifies the type of    */
/*                    the transaction parameter.  Its value will be  */
/*                    either '00EE'X or '00BB'X.                     */
/*                                                                   */
/*   The following parameters of CNMEGTP: GPCURSOR and GPNAMEH are   */
/*   not used when getting parameter by name.  As a result, they     */
/*   will NOT be initialized here.                                   */
/*********************************************************************/
GET_NAME: PROC;

  /*******************************************************************/
  /* Initialize input parameters to CNMEGTP.                         */
  /*******************************************************************/
  GPMETHOD = 'P';
  GPNAMEP  = 'Array_parameter_name';/* Parameter name whose value is*/
                                    /*  retrieved                   */
  GPINDEX  = 25;                   /*Indicates that the parameter is*/
                                   /* the 25th element of an array. */
  GPDATLEN = 256;                  /* Since GPDATA is defined as    */
                                   /* CHAR(256) VARYING             */
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 13 of 17)

```
CALL CNMEGTP (BHLBPTR,
              GPMETHOD,
              GPNAMEH,
              GPNAMEP,
              GPINDEX,
              GPDATA,
              GPDATLEN,
              GPTYPE,
              GPCURSOR);

/*****************************************************************/
/* Process the output -- This is only a skeleton.  Those return  */
/* codes that are NOT mentioned below, are unlikely to occur.    */
/*****************************************************************/
SELECT (HLBRC);
  WHEN (RET_GOOD)
    DO;
      /*************************************************************/
      /* Successful retrieval of transaction parameter             */
      /*************************************************************/
    END;

  WHEN (RET_BAD_LENGTH)
    DO;
      /*************************************************************/
      /* Re-check GPDATLEN                                         */
      /*************************************************************/
    END;

  WHEN (RET_BAD_NAME)
    DO;
      /*************************************************************/
      /* The name specified in GPNAMEP and its index does not      */
      /* exist in the parameter list.                              */
      /*************************************************************/
    END;

  WHEN (RET_NO_DATA)
    DO;
      /*************************************************************/
      /* The name specified by GPNAMEP and its index exists in     */
      /* the parameter list with no value.                         */
      /*************************************************************/
    END;
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 14 of 17)

```
    WHEN (RET_BAD_METHOD)
      DO;
        /*************************************************************/
        /* Re-check GPMETHOD.  Valid values are 'H', 'P' and 'N'.    */
        /*************************************************************/
      END;

    WHEN (RET_DATA_TRUNC)
      DO;
        /*************************************************************/
        /* GPDATA is not large enough to contain the value of the    */
        /* transaction parameter.                                    */
        /*                                                           */
        /* If user wishes to retrieve the current truncated          */
        /* parameter value again, he/she must perform the following */
        /* steps:                                                    */
        /*                                                           */
        /* 1. Save the name of the parameter and its index, GPNAMEP */
        /*    and and GPINDEX respectively.                          */
        /* 2. Allocate more space for GPDATA.  The additional        */
        /*    amount of space needed is equal to the difference      */
        /*    between HLBLENG and GPDATLEN.                          */
        /* 3. Update GPDATLEN to reflect GPDATA's new length.        */
        /* 4. Call CNMEGTP, with GPMETHOD of 'P' with the saved      */
        /*    GPNAMEP and GPINDEX (from step 1).                     */
        /* 5. Process the output as usual.                           */
        /*************************************************************/
      END;
  END;                              /* Select on HLBRC              */
END GET_NAME;
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 15 of 17)

```
/*********************************************************************/
/* SEND_REPLY procedure                                              */
/*   This procedure illustrates the correct process for sending a    */
/* reply to the NetView requester using the CNMESTR service routine.*/
/*                                                                   */
/* 1. Interchange the following parameters to reflect the correct    */
/*    sender and receiver:  Orignet, Origdom, Origtask and Destnet,  */
/*    Destdom, Desttask.  This swapping of destination and origin    */
/*    parameters will NOT be shown here.                             */
/*                                                                   */
/* 2. If the TRTYPE (received from the request) is QALL, a reply     */
/*    must be sent.  If the TRTYPE is QERR, any error resulting      */
/*    from accessing the database has to be sent in a reply.         */
/*********************************************************************/

SEND_REPLY: PROC;

  IF ((TRANS_TYPE = 'QALL') |
     ((TRANS_TYPE = 'QERR') &
      (RETURN_CODE ¬= RET_GOOD))) THEN /* Return code from          */
                                       /* accessing the database    */
    DO;
      /*************************************************************/
      /* Interchange origin and destination of netid, domain name  */
      /* and task name.  The codes are NOT shown here.             */
      /*************************************************************/

      /*************************************************************/
      /* After getting the VOCAB, CORR, and RESPCODE which are     */
      /* relevant to your application, you may want to form the    */
      /* parameter list (a singly linked list).  If there is no    */
      /* parameter list, you should initialize the parameter PARMS */
      /* to Null.  This code is NOT shown here.                    */
      /*************************************************************/
      CALL CNMESTR (BHLBPTR, TRANSID, TRTYPE, DESTNET,
                    DESTDOM, DESTTASK, ORIGNET, ORIGDOM,
                    ORIGTASK, VOCAB, CORR, RESPCODE, PARMS);
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 16 of 17)

```
      /*************************************************************/
      /* Process the return code -- This is only a skeleton.  Those */
      /* return codes that are NOT mentioned below, are unlikely to */
      /* occur.                                                    */
      /*************************************************************/
      SELECT (HLBRC);
        WHEN (RET_GOOD)
          DO;
            /*********************************************************/
            /* The reply gets put in the queue successfully.         */
            /*********************************************************/
          END;

        WHEN (RET_BAD_TRAN_HDR)
          DO;
            /*********************************************************/
            /* Check all header parameters.  Make sure that they     */
            /* are left justified, padded with blanks, etc..         */
            /*********************************************************/
          END;

        WHEN (RET_TRAN_REPLY_TRUNC)
          DO;
            /*********************************************************/
            /* The reply is longer than 31K bytes, and it gets       */
            /* truncated before sending.                             */
            /*********************************************************/
          END;

        WHEN (RET_PPI_FAILURE)
          DO;
            /*********************************************************/
            /* Check to make sure PPI has been defined correctly     */
            /* between NetView and the database server.              */
            /*********************************************************/
          END;

        WHEN (RET_BAD_TRAN_PARM)
          DO;
            /*********************************************************/
            /* Check the nth parameter of the parameter list to      */
            /* ensure that the parameter name length and/or the      */
            /* parameter value length is defined correctly.          */
            /*********************************************************/
          END;

       END;                          /* Select on HLBRC              */
     END;
 END SEND_REPLY;

 END SERTRAN;
```

Figure 13. Example of a PL/I Program Using the Server Support API (Part 17 of 17)

Example C Program Using the Server Support API

Figure 14 on page 42 is an example of a C program that uses the server support API. You can use this as a model for creating your own C program.

```
/*********************************************************************/
/*                                                                   */
/* C DRIVER  --  A database server written in C that drives the      */
/*               NetView/Bridge server support API.                  */
/*                                                                   */
/*********************************************************************/

/*********************************************************************/
/*                                                                   */
/*  #include files needed                                           */
/*  ----------------------                                           */
/*                                                                   */
/*  The following standard #include files are needed to compile     */
/*  this program:                                                    */
/*          stdio      string     stdefs                             */
/*          stdarg     stdlib     stdlib                             */
/*                                                                   */
/*  The following NetView/Bridge #include files are also needed.     */
/*  These are included by dsibc.                                     */
/*      dsicprm    - defines the parameter data structure           */
/*      dsibchlb   - defines the HLB data structure                 */
/*      dsibccnm   - defines the return codes for the Server        */
/*                   Support API routines                           */
/*      dsibccal   - function and macro call definitions for the    */
/*                   Server Support API                             */
/*      dsicvarc   - defines the varying-length character string    */
/*                                                                   */
/*********************************************************************/
#pragma runopts (NOEXECOPS,ISASIZE(4K),ISAINC(4K))

/*********************************************************************/
/* Standard C language include files                                 */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdefs.h>
#include <stdlib.h>
#include <stdarg.h>

#include "dsibc.h"                /* Includes NetView/Bridge files */

#define   NOT_AN_INDEX  -32767
#define   MAX_PARMS     200       /* Maximum number of parameters  */
#define   EBCDIC        0x00EE
#define   BINARY        0x00BB
```

Figure 14. Example of a C Program Using the Server Support API (Part 1 of 10)

```
/*********************************************************************/
/* ECB constants                                                     */
/*********************************************************************/
#define   ENDLIST       0x80000000 /* Mark end of list              */
#define   POSTED        0x40000000 /* Was the ECB posted            */

Dsibhlb *Bhlbptr;                  /* Pointer to the Server Support
                                      API HLB                       */

/*********************************************************************/
/* ECB constants                                                     */
/*********************************************************************/
void Gethdr();                     /* Get header routine            */
void getecb(long *ecbaddr, char *jobname); /* Get communications
                                              ECB routine           */


/*********************************************************************/
/* Define a 32 byte varying length character string                  */
/*********************************************************************/
typedef struct {
  short size;
  char  buffer??(32??);
              } D32varch;

/*********************************************************************/
/*                         M A I N L I N E                           */
/*********************************************************************/
main()
{
  /*******************************************************************/
  /* Parameters                                                      */
  /*******************************************************************/
  char     tiinq??(9??), tioutq??(9??);      /* DTS queues names    */
  int      tiinqlim;                 /* Input queue limit           */
  long     *waecbp;                  /* Pointer to ECB's            */
  long     ecbaddr,ecblist,ecb;      /* Event control block, list   */

  Dsicprm  Parm??(MAX_PARMS??);      /* Array of parameter blocks   */
  Dsicprm  *Parms;                   /* Pointer to first parm       */
  Dsicprm  *Curr;                    /* Pointer to current parm     */
  char     Parmlist??(MAX_PARMS??)??(256??); /* Array of parameter
                                                values              */
```

Figure 14. Example of a C Program Using the Server Support API (Part 2 of 10)

```
/*****************************************************************/
/* Variables for CNMESTR invocation                              */
/*****************************************************************/
char     Transid??(9??);
char     Trtype??(5??);
char     Vocab??(9??);
char     Corr??(9??);
char     Destnet??(9??);
char     Desttask??(9??);
char     Destdom??(9??);
char     Orignet??(9??);
char     Origtask??(9??);
char     Origdom??(9??);
char     Respcode??(9??);

/*****************************************************************/
/* Variables for CNMGETP invocation                              */
/*****************************************************************/
char     gpmethod;
char     gpnameh??(9??);
D32varch gpnamep;
int      gpindex;
Dsivarch gpdata;
int      gpdatlen;
short    gptype;
int      gpcursor;

/*****************************************************************/
/* Miscellaneous variables                                       */
/*****************************************************************/
int quiesced,emptyq,cleanq;/* Booleans                           */
int rc,gpcount,maxp,actlen;
char jobname??(8??);

quiesced = 0;                   /* Has Cnmetqu been invoked?     */
emptyq   = 0;                   /* Has queue been emptied yet?   */
cleanq   = 0;                   /* Has queue been cleaned yet?   */

/*****************************************************************/
/* Initialize the target database interface here !!              */
/* (Code is NOT supplied)                                        */
/*****************************************************************/
```

Figure 14. Example of a C Program Using the Server Support API (Part 3 of 10)

```
/*****************************************************************/
/* Initialize the NetView/Bridge Server Support API              */
/*****************************************************************/
strcpy(tiinq,"CINPUTQ1");        /* Not previously defined       */
strcpy(tioutq,"OUTPUTQ1");       /* Output queue in RTRINIT      */
tiinqlim = 4;                    /* 4 x # of ready tokens        */

Cnmetin(tiinq,tiinqlim,tioutq);

/*****************************************************************/
/* Loop forever - Exit when a QUIESCE is received                */
/*****************************************************************/
for (;;) {
  /***************************************************************/
  /* Note : The ready tokens on the queue upon initialization    */
  /*        will be handled prior to issuing more ready tokens.  */
  /***************************************************************/
  /* Get Next Transaction Request                                */
  /* A get transaction with an empty queue issues RET_NOT_FOUND. */
  /* In this case, emptyq is set to true, we will issue a ready  */
  /* token, wait for a transaction, and receive it when it comes.*/
  /* If the queue starts off non-empty, we don't want to issue a */
  /* ready token.  The transaction will be received and emptyq   */
  /* is set to false.  All transactions on the queue will be     */
  /* processed prior to issuing a ready token.                   */
  /***************************************************************/

  /***************************************************************/
  /* Get next transaction request before ready and wait          */
  /***************************************************************/

  /***************************************************************/
  /* Do the following until the queue is cleaned out             */
  /* If a transaction is found, skip the get transaction section,*/
  /* process the current transaction and loop back to here.      */
  /***************************************************************/
  if (!cleanq) {
    Cnmentr();
    rc = Bhlbptr->Hlbrc;
    if (rc == RET_GOOD)
      emptyq = 0;
    else
      emptyq = 1;
  }
```

Figure 14. Example of a C Program Using the Server Support API (Part 4 of 10)

```
/*************************************************************/
/* Do standard ready and wait before get transaction         */
/*************************************************************/
if (emptyq) {

  /* Upon entering this if, we know the queue has been cleaned  */
  cleanq = 1;

  /*************************************************************/
  /* Ready for next transaction request                        */
  /*  Issue a ready if not already in quiesced state           */
  /*************************************************************/
  if (!quiesced)
    Cnmertr();

  /*************************************************************/
  /* Wait for transaction request or ECB posting               */
  /*************************************************************/

  /*************************************************************/
  /* A stop command issued from the console will post the      */
  /* communications ECB causing the C Driver to terminate.     */
  /*************************************************************/

  /*************************************************************/
  /* Set up correct waecbp                                     */
  /* Get ecb address and jobname from assembler program        */
  /*************************************************************/
  getecb(&ecbaddr,jobname);
  jobname??(7??) = '\0';
  ecb = *((long *) ecbaddr);
  ecblist = ecbaddr;           /* ecblist has addr of ecb      */
  ecblist = ecblist | ENDLIST; /* Set 0 bit to 1               */
  waecbp = &ecblist;
  Cnmewat(&waecbp);
  rc=Bhlbptr->Hlbrc;
```

Figure 14. Example of a C Program Using the Server Support API (Part 5 of 10)

```
  /*************************************************************/
  /* Case 1 - User ECB posted (Quiesced from console           */
  /* Issue quiesce and wait for transaction                    */
  /*************************************************************/
  if (rc == RET_WAIT_POSTED) {
    quiesced = 1;
    Cnmetqu();
    waecbp = NULL;
    Cnmewat(&waecbp);
    rc=Bhlbptr->Hlbrc;
  }
  else
    if (rc == RET_WAIT_POSTED_PPI) {
      ecb = *((long *) ecbaddr);

    /*************************************************************/
    /* Case 2 - User ECB and PPI ECB posted                      */
    /* Issue quiesce and process transaction                     */
    /*************************************************************/
    if (ecb & POSTED) {
      Cnmetqu();
      quiesced = 1;
    }

    /*************************************************************/
    /* Case 3 - PPI ECB posted only                              */
    /* Process transaction                                       */
    /*************************************************************/

  }

  /*************************************************************/
  /* Get transaction request                                   */
  /*************************************************************/
  Cnmentr();

}                             /* End of if (emptyq)            */
```

Figure 14. Example of a C Program Using the Server Support API (Part 6 of 10)

```
/*************************************************************/
/* Get header parameters                                     */
/*************************************************************/
Gethdr("TRANSID ",Transid,8);
Gethdr("TRTYPE  ",Trtype,4);
Gethdr("CORR    ",Corr,8);
Gethdr("VOCAB   ",Vocab,8);
Gethdr("DESTNET ",Destnet,8);
Gethdr("DESTDOM ",Destdom,8);
Gethdr("DESTTASK",Desttask,8);
Gethdr("ORIGNET ",Orignet,8);
Gethdr("ORIGDOM ",Origdom,8);
Gethdr("ORIGTASK",Origtask,8);

/*************************************************************/
/* If transid = CNMQUISE then                                */
/*     1) If Origtask is this server's TIINQ name            */
/*              and in a quiesced state                      */
/*              then send terminate.                         */
/*     2) If Origtask is this server's TIINQ name            */
/*              and not in a quiesced state                  */
/*              then ignore.                                 */
/*************************************************************/
if (!strcmp(Transid,"CNMQUISE")) {  /* transid is Quiesce    */

  /***********************************************************/
  /* Terminate if Origtask is this server's TIINQ name       */
  /* and server is in a quiesced state                       */
  /***********************************************************/
  if ((!strcmp(Origtask,tiinq)) && (quiesced)) {
    Cnmetrm();                /* Terminate Server Support API  */
    exit(999);
  }                           /* Quiesced by autotask          */
}                             /* Transid of QUIESCE            */

else {                        /* Other Transid                 */
  gpmethod = 'N';
  gpdatlen = 256;
  gpcursor = 0;
  gpcount  = 0;
  Cnmegtp(&gpmethod,gpnameh,&gpnamep,&gpindex,
          &gpdata,gpdatlen,&gptype,&gpcursor);
  rc=Bhlbptr->Hlbrc;

  while (((rc==RET_GOOD) || (rc==RET_DATA_TRUNC))
                        && (gpcount < maxp)) {
    actlen = Bhlbptr->Hlbleng;
```

Figure 14. Example of a C Program Using the Server Support API (Part 7 of 10)

```
  /*************************************************************/
  /* On return from Cnmegtp, Bhlbptr->Hlbleng contains the    */
  /* length of the data.  Gpdatlen contains the length of the */
  /* supplied buffer.  If the return code is RET_DATA_TRUNC   */
  /* the server may use the truncated length of the data,     */
  /* which is equal to the buffer length.                     */
  /*************************************************************/
  if (rc == RET_DATA_TRUNC)
    actlen = gpdatlen;

  /*************************************************************/
  /* Alternately, and not shown here, the server may recall   */
  /* the gpnamep, gptype, and gpindex.  With this information */
  /* Cnmegtp by name may be called with an adequate buffer.   */
  /*************************************************************/

  gpnamep.buffer??(gpnamep.size??) = '\0';
  gpdata.buffer??(actlen??) = '\0';

  strcpy(Parm??(gpcount??).prmname,gpnamep.buffer);
  Parm??(gpcount??).prmnaml = gpnamep.size;
  Parm??(gpcount??).prmindex = gpindex;
  Parm??(gpcount??).prmtype = gptype;
  strcpy(Parmlist??(gpcount??),gpdata.buffer);
  Parm??(gpcount??).prmptr = Parmlist??(gpcount??);
  Parm??(gpcount??).prmleng = actlen;
  if (gpcount < maxp)
    Parm??(gpcount??).prmlink = &Parm??(gpcount+1??);
  gpcount++;
  Cnmegtp(&gpmethod,gpnameh,&gpnamep,&gpindex,
          &gpdata,gpdatlen,&gptype,&gpcursor);
}

Parm??(gpcount-1??).prmlink = NULL;
```

Figure 14. Example of a C Program Using the Server Support API (Part 8 of 10)

```
      /*************************************************************/
      /* Process transactions                                      */
      /* Make calls to routines to interact with database or other  */
      /* server.  Send in parameter list on input.  Set Respcode    */
      /* and parameter list on output from these routines.          */
      /* Following are sample invocations of server defined         */
      /* processing routines.                                      */
      /*************************************************************/
      if (!strcmp(Transid,"CREATE"))
        Create(Respcode,Parm);
      else
        if (!strcmp(Transid,"SEARCH"))
          Search(Respcode,Parm);
        else
          if (!strcmp(Transid,"UPDATE"))
            Update(Respcode,Parm);

      /*************************************************************/
      /* Send the transaction reply                                */
      /*  (Note that the origin and destination parameters have    */
      /*  been interchanged)                                       */
      /*************************************************************/

      /*************************************************************/
      /* If Trtype = QERR and Respcode != 0 or if Trtype = QALL,    */
      /* we will return the message, else continue to next          */
      /*************************************************************/
      if (((!strcmp(Trtype,"QERR")) && (strcmp(Respcode,
      "00000000")))
                || (!strcmp(Trtype,"QALL"))) {

        strcpy(Trtype,"RPLY");
        Cnmestr(Transid,Trtype,Orignet,Origdom,Origtask,
                Destnet,Destdom,Desttask,Vocab,Corr,Respcode,&Parms);;
      }                             /* End of Send Reply Loop        */
    }                               /* End of Transid not Quiesce    */
    cleanq = 0;
  }                                 /* End of Processing Loop        */
}                                   /* End of Main                   */
```

Figure 14. Example of a C Program Using the Server Support API (Part 9 of 10)

```
/*********************************************************************/
/* GetHdr ()                                                         */
/* Function:  To get header parameters                               */
/*********************************************************************/
void Gethdr(char *gpnameh,char *header,int gpdatlen)

{
  char     gpmethod;
  D32varch gpnamep;
  int      gpindex,gpcursor;
  Dsivarch gpdata;
  short    gptype;
  int      rc,actlen;

  gpmethod = 'H';
  Cnmegtp(&gpmethod,gpnameh,&gpnamep,&gpindex,
          &gpdata,gpdatlen,&gptype,&gpcursor);
  actlen = Bhlbptr->Hlbleng;
  if (rc == RET_DATA_TRUNC)
    actlen = gpdatlen;
  gpdata.buffer??(actlen??) = '\0';
}                                  /* End of Gethdr                  */
```

Figure 14. Example of a C Program Using the Server Support API (Part 10 of 10)

Assembler Source Code for Obtaining the Address of the Communications ECB

Figure 15 on page 52 is an example of an assembler program that you can use to obtain the address of the communications event control block (ECB):

```
******************************** TOP OF DATA  ************************
         TITLE 'GETECB - GET JOBNAME AND COMMUNICATIONS ECB ADDR'
*********************************************************************
*
* FUNCTION:
*    THIS MODULE PASSES TO THE CALLER THE JOBNAME OF THIS JOB/TASK
*    AND THE ADDRESS OF THE COMMUNICATIONS ECB FOR STOP/MODIFY
*    COMMANDS.
*
* NOTES:
*    REGISTER CONVENTIONS:  STANDARD MVS/370 LINKAGE CONVENTIONS
*
* MODULE CHARACTERISTICS:
*    PROCESSOR:  ASSEMBLER H VERSION 2
*    TYPE:       PROCEDURE
*    SIZE:       SEE ASSEMBLER LISTING
*    ATTRIBUTES: RENT,REUS,AMODE ANY,RMODE ANY
*
* PROCEDURE INVOCATION: CALL GETECB(ECB_ADDRESS,JOB_NAME);
*
*
*  INPUT: R1-> ADDRESS OF PARAMETER LIST
*              +0 ADDRESS OF ADDRESS OF COMMUNICATIONS ECB
*              +4 ADDRESS OF JOBNAME
*
*         R12-> PL/I TCA (NOT MODIFIED)
*         R13-> REGISTER SAVE AREA
*         R14=  RETURN ADDRESS
*         R15=  ENTRY ADDRESS
*
*
* OUTPUT:
*         R1-> ADDRESS OF PARAMETER LIST
*              +0 ADDRESS OF ADDRESS OF COMMUNICATIONS ECB
*              +4 ADDRESS OF JOBNAME
*
*    RETURN CODES - NONE.
*
*
* MACROS:
*    EXECUTABLE:
*       SAVE
*       GETMAIN
*       FREEMAIN
*       EXTRACT
*       RETURN
```

Figure 15. Assembler Code for Obtaining the Address of the Communications ECB (Part 1 of 4)

```
*    NON-EXECUTABLE:
*       IEZCOM           - MAP COMM AREA
*       IEFTIOT          - MAP TASK INPUT OUTPUT TABLE
*
*********************************************************************
GETECB CSECT
GETECB AMODE ANY
GETECB RMODE ANY
         SPACE
         USING GETECB,R15
         B     AFTRID
         DC    C'GETECB &SYSDATE &SYSTIME'
         SPACE
AFTRID   DS    0H
         SAVE  (14,12)
         DROP  R15
         LR    R11,R15            LOAD BASE REGISTER
         USING GETECB,R11         SET UP CSECT ADDRESSABILITY
         LR    R3,R1              SAVE ADDRESS OF PARM LIST
         GETMAIN RU,LV=WORKLEN,LOC=BELOW  SAVE AREA AND WORK AREA
         ST    R13,4(,R1)
         ST    R1,8(,R13)
         LR    R13,R1
         USING WORKAREA,R13
         SPACE
         MVC   EXTRA(EXTLEN),EXTRACT    MOVE EXTRACT TO BELOW LINE
         EXTRACT TOOTH,'S',FIELDS=(TIOT,COMM),MF=(E,EXTRA)
         SPACE
         L     R5,COMMADDR     GET ADDRESS OF COMMUNICATIONS AREA
         USING COMLIST,R5
         L     R6,COMECBPT     GET ADDRESS OF COMM ECB
         L     R4,0(,R3)       GET FIRST PARAMETER ADDRESS
         ST    R6,0(,R4)       STORE COMM ECB ADDRESS IN FIRST PARM
         SPACE
         L     R5,TIOTADDR     GET ADDRESS OF TIOT
         USING TIOT,R5
         L     R4,4(,R3)        GET SECOND PARAMETER ADDRESS
         MVC   0(8,R4),TIOCNJOB  MOVE JOBNAME TO SECOND PARAMETER
         SPACE
EXIT     EQU   *
         LR    R10,R13
         L     R13,4(,R13)
         FREEMAIN RU,LV=WORKLEN,A=(R10)
         RETURN (14,12)
         EJECT
```

Figure 15. Assembler Code for Obtaining the Address of the Communications ECB (Part 2 of 4)

```
EXTRACT  EXTRACT MF=L
EXTLEN   EQU   *-EXTRACT
         LTORG
WORKAREA DSECT
SAVEAREA DS    20F
EXTRA    DS    XL(EXTLEN)
TOOTH    DS    0F
TIOTADDR DS    A
COMMADDR DS    A
WORKLEN  EQU   *-WORKAREA
         IEZCOM
TIOT     DSECT
         IEFTIOT1
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
```

Figure 15. Assembler Code for Obtaining the Address of the Communications ECB (Part 3 of 4)

```
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END   GETECB
```

Figure 15. Assembler Code for Obtaining the Address of the Communications ECB (Part 4 of 4)

Chapter 4. Writing a NetView Bridge Command Procedure Using the NetView Bridge Requestor API

This chapter describes general-use programming interface and associated guidance information.

This chapter describes how to write command procedures using REXX or NetView command list language, and another set of procedures for writing command procedures using PL/I or C.

If you are not familiar with writing command procedures for the NetView program, refer to *Tivoli NetView for OS/390 Customization: Using REXX and the NetView Command List Language* , or *Tivoli NetView for OS/390 Customization: Using PL/I and C*.

Understanding Transaction Size Limitations

A total of 31K bytes is available for a single transaction, which includes all user data combined with packaging code that must accompany your parameters, but excludes header information.

Note: The actual limit to the size of a transaction is slightly greater than 31K. However, staying within the 31K limit prevents error conditions or truncated replies.

Header Information

Header information is data that is automatically included in every transaction, such as:

- Correlator information
- Destination information
- Origination information
- Transaction ID
- Transaction type
- Vocabulary parameter

Header information does not count toward the allowed maximum of 31K bytes.

User Data

User data consists of the many parameters that are used by your database server. This is the server-specific information that is passed in transactions.

Packaging Code

Packaging code envelopes the parameter information passed by NetView Bridge. There are 23 bytes of packaging code attached to each parameter.

Size of a Parameter

Each parameter consists of a name, type, and value. The method for calculating the size of a packaged parameter is:

```
23 bytes + length of parameter name + length
of  parameter value = size of the packaged parameter
```

Total Size of a Transaction

To determine the **total** size of a transaction, calculate the packaged parameter length of **each** parameter in a transaction and add the results.

Note: The total size must be less than 31K. If the size of a transaction request exceeds 31K, the NetView Bridge Requester API rejects it. If the size of a transaction reply exceeds 31K, it is truncated to 31K, and forwarded. However, the program that sends the reply receives a nonzero return code.

Writing Command Procedures Using REXX or the NetView Command List Language

This section describes the procedures for creating a command procedure to work with NetView Bridge using either the REXX or NetView command list language. Examples of command procedures are included as models for creating a command procedure. See Figure 16 on page 58 and Figure 17 on page 62 for more information.

The following steps, along with Figure 16 and Figure 17, assume that the command procedure is being run under an operator station task (OST). If you want to write command procedures to run under the primary program operator interface task (PPT), you must use the NetView automation table to capture the NetView Bridge message DWO548I and drive another command procedure to process the reply. Refer to *Tivoli NetView for OS/390 Automation Guide* for a description of this procedure.

Note: Running under the PPT is not recommended because you cannot wait for transaction replies. The PPT does not support waits.

Complete the following steps in order.

Step 1. Initializing Parameters

At the beginning of your command procedure, assign values to initialize all parameters for the transactions you want to execute. See “TRANSND—Send Transaction Request to Database Server” on page 91 for complete details on the parameters, their usage, and syntax.

See your server documentation for details on any parameters your server may require to process your transaction.

Step 2. Preparing to Receive the Reply

Write the lines of code to prepare for receiving the reply.

If you are writing this transaction using REXX, issue the NetView TRAP command (see Figure 16 on page 58). This traps message DWO548I. (Message DWO548I is the first line of the two-line message used to send a transaction reply back to the NetView Requester API requesting command procedure.)

If you are writing this transaction using the NetView command list language, issue the &WAIT command (see Figure 17 on page 62). This causes your command procedure to wait on the receipt of message DWO548I.

Step 3. Sending the Transaction Request

Write the lines of code to issue the NetView Bridge TRANSND command. See “TRANSND—Send Transaction Request to Database Server” on page 91 for more information on the TRANSND command.

You can send transactions across networks or domains by changing DESTNET and DESTDOM to the remote network name or remote domain name of your choice, respectively.

Note: For a NetView command list language command procedure, this command is already part of the &WAIT statement.

Step 4. Processing the Reply

Write the lines of code to process the reply.

If you are writing this transaction using REXX, issue the NetView MSGREAD command to read the trapped message DWO548I from the message queue, followed by the TRANRCV command to process the reply parameters (see Figure 16 on page 58).

If you are writing this transaction using the NetView command list language, issue the TRANRCV command (see Figure 17 on page 62).

With the REXX or NetView command list language, the TRANRCV command creates local variables into which the reply data is placed. See “TRANRCV—Receive a Transaction Reply” on page 94 for a list of these variables.

At this point, you can also write the necessary code to process the reply data in any manner you want.

Example of a Command Processor Using REXX

Figure 16 on page 58 is an example of a REXX EXEC that uses the TRANSND and TRANRCV commands. You might want to use this example as a model for creating your own command processor.

```
/*********************************************************************/
/* The following is a sample of a REXX program, designed to utilize  */
/* the functions of the NetView Bridge.                              */
/*********************************************************************/

/*********************************************************************/
/* Set parameter values                                              */
/*********************************************************************/
PARM1='VALUE1'
PARM2='VALUE2'
MULTI_VALUE_PARM.0=2               /* Number of elements in array    */
MULTI_VALUE_PARM.1='THIS IS A SAMPLE OF USING AN ARRAY'
MULTI_VALUE_PARM.2='TO SPECIFY A MULTI-VALUE PARAMETER.'

/*********************************************************************/
/* Set TRANSND values                                                */
/*********************************************************************/
TRANSID='CREATE'                   /* Transaction name               */
TRTYPE='QALL'                      /* Transaction type               */
DESTNET='NETA'                     /* Destination network name       */
DESTDOM='CNM01'                    /* Destination domain name        */
DESTTASK='BRDGDSP1'                /* BRIDGE dispatcher name         */
VOCAB='TABLE1'                     /* VOCAB table name               */
CORR='CORRVAL'                     /* Correlator value               */

PARMVAR='PARM1 PARM2 MULTI_VALUE_PARM. '

/*********************************************************************/
/* Prepare for reply, then request transaction execution             */
/*********************************************************************/
'TRAP AND SUPPRESS ONLY MESSAGES DWO548I'
'TRANSND TRANSID='TRANSID',TRTYPE='TRTYPE',DESTNET='DESTNET,
          'DESTDOM='DESTDOM',DESTTASK='DESTTASK',VOCAB='VOCAB,
          'CORR='CORR',PARMVAR=PARMVAR'

If rc ¬= 0 Then                    /* TRANSND command failed         */
  Call EXIT_ROUTINE TRSNDERR rc
Else                               /* TRANSND command rc = 0         */
  Do
    /*****************************************************************/
    /* Process all DWO548I messages.  Loop until we receive a        */
    /* matching RPCORREL value, or until we EXIT due to an error.    */
    /*****************************************************************/
    say 'TRANSND command rc = 0 -- Waiting for reply...'
    FLAG = 0
    Do until FLAG = 1
      Call DWOPROC
      Call PRINTOUT
```

Figure 16. Example of a REXX EXEC Using TRANSND and TRANRCV (Part 1 of 4)

```
      /*****************************************************************/
      /* Determine whether this is the reply that I am waiting for.    */
      /* CORREL is a system assigned variable, which is set at         */
      /* TRANSND execution time.                                       */
      /*****************************************************************/
      If RPCORREL ¬= CORREL Then    /* Wrong DWO548I message received */
        say 'Correlators do not match...'
      Else
        FLAG = 1
    End
  End                               /* End else                      */

Call EXIT_ROUTINE NOERR

/*********************************************************************/
/* DWOPROC procedure to WAIT on messages                             */
/* DWOPROC will wait 2 minutes for DWO548I. After timeout, it will   */
/* exit in error. If successful, it will issue TRANRCV.              */
/* If the return code from TRANRCV ¬= 0, we will exit.  Otherwise,   */
/* we will return to the main body of the exec.                      */
/*********************************************************************/
DWOPROC:

'WAIT 2 MINUTES FOR MESSAGES '
Select
  When EVENT() = 'M' Then
    Do
      say 'TRANSND reply(DWO548I) received...'
      'MSGREAD'
      'TRANRCV PARMVAR=TRRCVVAR'    /* Request reply parameters       */
      Select
        When rc = 2 Then
          say 'Parm name or value truncated -- Processing continues...'
        When rc = 0 Then
          say 'TRANRCV command rc = 0 -- Processing continues...'
        Otherwise
          Call EXIT_ROUTINE TRRCVERR rc
      End                           /* End select                    */
    End
  When EVENT() = 'T' Then
    Call EXIT_ROUTINE TMOUTERR
  When EVENT() = 'G' Then
    Call EXIT_ROUTINE OPGOERR
  Otherwise
    Call EXIT_ROUTINE EVENTERR
End                                 /* End select                    */
Return                              /* End DWOPROC                   */
```

Figure 16. Example of a REXX EXEC Using TRANSND and TRANRCV (Part 2 of 4)

```
/*********************************************************************/
/* Printout will display all header and non-header information to the */
/* operator.                                                         */
/*********************************************************************/
PRINTOUT:

/*********************************************************************/
/* Print header information                                          */
/*********************************************************************/
say 'RPTRANSID  = 'RPTRANSID
say 'RPTRTYPE   = 'RPTRTYPE
say 'RPDESNET   = 'RPDESNET
say 'RPDESDOM   = 'RPDESDOM
say 'RPDESTSK   = 'RPDESTSK
say 'RPORIGNET  = 'RPORIGNET
say 'RPORIGDOM  = 'RPORIGDOM
say 'RPORIGTSK  = 'RPORIGTSK
say 'RPCORREL   = 'RPCORREL
say 'RPVOCAB    = 'RPVOCAB
say 'RPRESPCODE = 'RPRESPCODE

/*********************************************************************/
/* Print non-header information                                      */
/*********************************************************************/
Do i = 1 to words(TRRCVVAR)         /* Loop through list of keywords  */
  /*******************************************************************/
  /* Print each keyword, and its value                               */
  /*******************************************************************/
  say word(TRRCVVAR,i)' = 'value(word(TRRCVVAR,i))
End
Return                              /* End PRINTOUT                   */
```

Figure 16. Example of a REXX EXEC Using TRANSND and TRANRCV (Part 3 of 4)

```
/*********************************************************************/
/* Exit routine                                                      */
/*********************************************************************/
EXIT_ROUTINE:

arg ERRMSG ERRCODE
Select
  When ERRMSG = 'TRSNDERR' Then
    say 'TRANSND failed with rc = 'ERRCODE
  When ERRMSG = 'TRRCVERR' Then
    say 'TRANRCV failed with rc = 'ERRCODE
  When ERRMSG = 'TMOUTERR' Then
    say 'No DWO548I message received...Timed-out after 2 minutes...'
  When ERRMSG = 'OPGOERR' Then
    say 'Operator entered "GO" command...Terminated "WAIT" state...'
  When ERRMSG = 'EVENTERR' Then
    say 'MSGREAD failure detected...'
  When ERRMSG = 'NOERR' Then
    say 'Program completed successfully...'
  Otherwise
    NOP
End                                 /* End select                    */

parse source . . prgname .          /* Get the program name          */
say prgname' complete.'
Exit
Return                              /* End EXIT_ROUTINE              */
```

Figure 16. Example of a REXX EXEC Using TRANSND and TRANRCV (Part 4 of 4)

Example of a Command Processor Using the NetView Command List Language

Figure 17 on page 62 is an example of a NetView command list that uses the TRANSND and TRANRCV commands. You might want to use this example as a model for creating your own command processor.

```
EXAMPLE CLIST
&CONTROL ERR
*********************************************************************
*  Sample NetView Command List Language transaction program         *
*********************************************************************

*********************************************************************
* Set parameter values                                              *
*********************************************************************
&PARM1 = 'VALUE1'
&PARM2 = 'VALUE2'

*********************************************************************
* Set TRANSND values                                                *
*********************************************************************
&TRANSID = 'CREATE'
&TRTYPE = 'QALL'
&DESTNET = 'NETA'
&DESTDOM = 'CNM01'
&DESTTASK = 'BRDGDSP1'
&VOCAB = 'TABLE1'
&CORR = 'CORRVAL'

&PARMVAR = 'PARM1 PARM2'

*********************************************************************
* Prepare for reply, then request transaction execution
*********************************************************************
&WAIT CONTWAIT SUPPRESS
&WRITE Sending transaction now...
&WAIT 'TRANSND TRANSID=&TRANSID,TRTYPE=&TRTYPE,DESTNET=
       &DESTNET,+
       DESTDOM=&DESTDOM,DESTTASK=&DESTTASK,VOCAB=&VOCAB,+
       CORR=&CORR,PARMVAR=PARMVAR' DWO548I=-DWOPROC +
       *120=-TMOUTERR *ERROR=-TRSNDERR *ENDWAIT=-OPGOERR

*********************************************************************
* DWOPROC                                                           *
*********************************************************************
-DWOPROC
&WRITE TRANSND reply (DWO548I) received...
TRANRCV PARMVAR=TRRCVVAR
&IF &RETCODE = 0 &THEN &GOTO -OKRC
&IF &RETCODE ¬= 2 &THEN &GOTO -TRRCVERR
&WRITE Parm name or value truncated
-OKRC
&WRITE TRANRCV command rc = &RETCODE    Processing continues...
```

Figure 17. Example of a NetView Command List using TRANSND and TRANRCV (Part 1 of 3)

```
*********************************************************************
* Display all header and non-header information to the operator     *
*********************************************************************
-PRINTOUT
*********************************************************************
* Print header information                                          *
*********************************************************************
&WRITE RPTRANSID  = &RPTRANSID
&WRITE RPTRTYPE   = &RPTRTYPE
&WRITE RPDESNET   = &RPDESNET
&WRITE RPDESDOM   = &RPDESDOM
&WRITE RPDESTSK   = &RPDESTSK
&WRITE RPORIGNET  = &RPORIGNET
&WRITE RPORIGDOM  = &RPORIGDOM
&WRITE RPORIGTSK  = &RPORIGTSK
&WRITE RPCORREL   = &RPCORREL
&WRITE RPVOCAB    = &RPVOCAB
&WRITE RPRESPCODE = &RPRESPCODE

*********************************************************************
* Print non-header information                                      *
*********************************************************************
&I = 1
&P = 0
&LEN = &LENGTH &TRRCVVAR
-PARMNUM
&P = &P + 1
&PARM&P = '
-REPLYLEN
&IF &I > &LEN &THEN &GOTO -CHEKCORR
&I = &I + 1
&CHAR = &SUBSTR &TRRCVVAR &I 1
&IF .&CHAR = . &THEN &GOTO -WRITPARM
&PARM&P = &CONCAT &PARM&P &CHAR
&GOTO -REPLYLEN
-WRITPARM
&WRITE PARM&P ==>  &PARM&P  is  &&PARM&P
&GOTO -PARMNUM
```

Figure 17. Example of a NetView Command List using TRANSND and TRANRCV (Part 2 of 3)

```
-CHEKCORR
*************************************************************************
* Determine whether this is the reply I am waiting for                  *
*************************************************************************
&IF &RPCORREL = &CORR &THEN &GOTO -DONE
&WRITE Correlators do not match...
&WAIT CONTINUE

-DONE
&WAIT ENDWAIT
GO
&WRITE Program completed successfully...
&GOTO -END

-OPGOERR
&WRITE Operator entered "GO" command... Terminated "WAIT" state...
&GOTO -END

-TRSNDERR
&WRITE TRANSND failed with rc = &RETCODE
&GOTO -END

-TMOUTERR
&WRITE No DWO548I message received... Timed out after 2 minutes...
&GOTO -END

-TRRCVERR
&WRITE TRANRCV failed with rc = &RETCODE
&GOTO -END

-END
&WRITE EXAMPLE complete
&EXIT
```

Figure 17. Example of a NetView Command List using TRANSND and TRANRCV (Part 3 of 3)

Writing Command Procedures Using PL/I or C

This section describes the procedures for creating a command procedure to work with NetView Bridge using either the PL/I or C programming language. Because VM and VSE do not support HLLs, such as PL/I and C, this section only applies to NetView Bridge running on MVS. Examples of command procedures are included as models for creating a command procedure (see Figure 18 on page 66 and Figure 19 on page 71).

The following steps, along with Figure 18 and Figure 19, assume that the command procedure is being run under an operator station task (OST). If you want to write command procedures to run under the primary program operator interface task (PPT), you must use the NetView automation table to capture the NetView Bridge message DWO548I and drive another command procedure to process the reply. See “The Automation Table” in *Tivoli NetView for OS/390 Automation Guide* for a description of this procedure.

Note: Running under the PPT is not recommended because you cannot wait for transaction replies. The PPT does not support waits.

Complete the following steps in order.

Step 1. Initializing Parameters and Using Transaction Parameter Blocks

The NetView program contains a parameter block structure specifically for use with the NetView Bridge Requester API service routine CNMSNDT (send a transaction request). The parameter block structure has two separate versions—one for PL/I (DSIPPRM) and one for C (DSICPRM).

When you write the code to initialize your parameters, place each nonheader parameter in its own allocated copy of the parameter block structure. Your code must then store the required information for this parameter in this parameter block structure.

Finally, write code to link these parameter block structures together for use by the CNMSNDT service routine. To do this, set the pointer field (PRMLINK) in each parameter block to link them all together. (See Figure 8 on page 21 for an example of how this is done.)

At the beginning of your command procedure, assign values to initialize the appropriate parameters for the transactions you want to execute. Refer to "CNMSNDT (CNMSENDTR)—Send Transaction Request to Database Server" on page 96 for complete details on the parameters, their usage, and syntax.

See your server documentation for details on any parameters your server may require to process your transaction.

Step 2. Preparing to Receive the Reply

Write the lines of code to prepare for receiving the reply.

To do this, invoke the NetView HLL service routine CNMCMD to issue the TRAP command. This traps message DWO548I. (Message DWO548I is the first line of the two-line message used to send a transaction reply back to the NetView Bridge Requester API requesting command procedure.) See Figure 18 on page 66 or Figure 19 on page 71 for an example of how you might write this code.

Step 3. Sending the Transaction Request

Write the lines of code to invoke the NetView Bridge service routine CNMSNDT (see Figure 18 and Figure 19). This service routine sends the transaction to the NetView Bridge Requester API.

Step 4. Processing the Reply

To obtain the reply data using HLL code, you must do two things:

1. Write the code to retrieve the header parameter (method H) using the NetView Bridge HLL service routine CNMGETP.
2. Write the code to retrieve the parameter information (method N or P) using the NetView Bridge HLL service routine CNMGETP.

See "CNMGETP (CNMGETPARM)—Get Transaction Reply Parameters" on page 100 for details on how to use this service routine.

At this point, you can also write the code necessary to process the reply data in any manner you want.

Step 5. Compiling and Link-Editing Your HLL Command Procedures

At this point, you need to compile and link-edit your HLL command procedures. See "Compiling, Link-Editing, and Running Your HLL Program" in *Tivoli NetView for OS/390 Customization: Using PL/I and C* for detailed instructions on this process.

Example of a Command Processor Using PL/I

Figure 18 is an example of a PL/I program that uses the NetView Bridge Requester API. You might want to use this example as a model for creating your own command processor.

Note: VM and VSE do not support PL/I or other HLLs.

```
PLISAMP: PROC(HLBPTR,CMDBUF,ORIGBLCK) OPTIONS(MAIN,REENTRANT);
/*********************************************************************/
/*                                                                   */
/*   Sample PL/I language transaction command processor              */
/*                                                                   */
/*   This sample command processor builds a transaction and          */
/*   sends it across the NetView bridge product. It is not           */
/*   written with any particular server or database in mind as       */
/*   all parameters are hard coded.                                  */
/*                                                                   */
/*   The following are demonstrated by this procedure:               */
/*     1. The correct use of CNMSNDT to send a transaction request   */
/*        over the bridge                                            */
/*     2. The procedure for WAITing and TRAPping on the reply        */
/*     3. The processing of the REPLY parameter using CNMGETP:       */
/*             CORR            Using CNMGETP method H                */
/*             REASONCODE      Using CNMGETP method P                */
/*             ALL OTHERS      Using CNMGETP method N                */
/*                                                                   */
/*********************************************************************/
%INCLUDE DSIPLI;                  /* DSIPLI includes required      */
                                  /* control blocks and include    */
                                  /* files                         */

DCL HLBPTR   PTR;                 /* Pointer to the HLB            */
DCL CMDBUF   CHAR(*) VARYING;     /* Command buffer                */
DCL ORIGBLCK CHAR(40);            /* Area for the ORIG block       */
DCL ORIGIN   PTR;                 /* Pointer to the ORIG block     */
DCL (ADDR,NULL) BUILTIN;
DCL LENGTH BUILTIN;

/*********************************************************************/
/* Declarations for CNMSNDT                                          */
/*********************************************************************/
DCL TRANSID  CHAR(8) INIT('CREATE  '); /* Transaction ID           */
DCL TRTYPE   CHAR(4) INIT('QALL');     /* Transaction type         */
DCL DESTNET  CHAR(8) INIT('NETA    '); /* Destination network name */
DCL DESTDOM  CHAR(8) INIT('CNM01   '); /* Destination domain name  */
DCL DESTTASK CHAR(8) INIT('BRDGDSP1'); /* Bridge dispatcher name   */
DCL VOCAB    CHAR(8) INIT('TABLE1  '); /* Vocab table name         */
DCL CORR     CHAR(8) INIT('CORRVAL '); /* Correlator value         */

DCL PARMLIST PTR;                 /* Pointer to parameter block    */
DCL STARTPTR PTR;                 /* Pointer to the beginning of   */
                                  /* the parameter list            */

DCL NOT_AN_INDEX  CHAR(6) INIT('-32767');
```

Figure 18. Example of a PL/I Program Using the NetView Bridge Requester API (Part 1 of 5)

```
/*********************************************************************/
/* Declarations for CNMGETP                                          */
/*********************************************************************/
DCL GPMETHOD CHAR(1);                     /* GET method (H,P,N)     */
DCL GPNAMEH  CHAR(8);                     /* Header name area       */
DCL GPNAMEP  CHAR(31) VARYING;            /* Parameter name area    */
DCL GPINDEX  FIXED BIN(31,0) INIT(0);     /* Index value            */
DCL GPDATA   CHAR(256) VARYING;           /* Data buffer area       */
DCL GPDATLEN FIXED BIN(31,0) INIT(256);   /* Max length of data     */
DCL GPTYPE   BIT(16) INIT('00EE'BX);      /* EBCDIC or BINARY       */
DCL GPQUEUE  FIXED BIN(31,0) INIT(TRAPQ); /* Data queue             */
DCL GPCURSOR FIXED BIN(31,0) INIT(0);     /* Cursor location in msg */

DCL P1NAM    CHAR(9)  INIT('PARMNAME1');
DCL P1VAL    CHAR(6)  INIT('VALUE1');
DCL P2NAM    CHAR(9)  INIT('PARMNAME2');
DCL P2VAL    CHAR(6)  INIT('VALUE2');
DCL P3NAM    CHAR(8)  INIT('MULTPARM');   /* P3NAM and P4NAM are    */
DCL P3VAL    CHAR(31) INIT('MULTVALUE1'); /* used to show how to    */
DCL P4NAM    CHAR(8)  INIT('MULTPARM');   /* use multi-line         */
DCL P4VAL    CHAR(31) INIT('MULTVALUE2'); /* parameters             */

/*********************************************************************/
/* Construct the linked list of parameters.  Allocate and            */
/* initialize the first parameter block.                             */
/*********************************************************************/
ALLOCATE DSIPPRM SET (PARMLIST);
STARTPTR = PARMLIST;                    /* Start of linked list      */
PARMLIST -> PRMINDEX = NOT_AN_INDEX;    /* Index set to -32767       */
PARMLIST -> PRMNAML  = LENGTH(P1NAM);   /* Length of 1st parm name   */
PARMLIST -> PRMNAME  = P1NAM;           /* 1st parm name             */
PARMLIST -> PRMTYPE  = PRM_TYPEEBDC;    /* 1st parm is in EBCDIC     */
PARMLIST -> PRMLENG  = LENGTH(P1VAL);   /* Length of 1st parm value  */
PARMLIST -> PRMPTR   = ADDR(P1VAL);     /* Address of 1st parm value */

/*********************************************************************/
/* Allocate and initialize second parameter block                    */
/*********************************************************************/
ALLOCATE DSIPPRM SET(PARMLIST ->PRMLINK);
PARMLIST = PARMLIST -> PRMLINK;         /* Link to 2nd parm in list  */
PARMLIST -> PRMINDEX = NOT_AN_INDEX;    /* Index set to -32767       */
PARMLIST -> PRMNAML  = LENGTH(P2NAM);   /* Length of 2nd parm name   */
PARMLIST -> PRMNAME  = P2NAM;           /* 2nd parm name             */
PARMLIST -> PRMTYPE  = PRM_TYPEEBDC;    /* 2nd parm is in EBCDIC     */
PARMLIST -> PRMLENG  = LENGTH(P2VAL);   /* Length of 2nd parm value  */
PARMLIST -> PRMPTR   = ADDR(P2VAL);     /* Address of 2nd parm value */
```

Figure 18. Example of a PL/I Program Using the NetView Bridge Requester API (Part 2 of 5)

```
/*********************************************************************/
/* Allocate and initialize third parameter block                     */
/*********************************************************************/
ALLOCATE DSIPPRM SET(PARMLIST ->PRMLINK);
PARMLIST = PARMLIST -> PRMLINK;        /* Link to 3rd parm in list  */
PARMLIST -> PRMINDEX = 1;              /* 1st line of multi-parm    */
PARMLIST -> PRMNAML  = LENGTH(P3NAM);  /* Length of 3rd parm name   */
PARMLIST -> PRMNAME  = P3NAM;          /* 3rd parm name             */
PARMLIST -> PRMTYPE  = PRM_TYPEEBDC;   /* 3rd parm is in EBCDIC     */
PARMLIST -> PRMLENG  = LENGTH(P3VAL);  /* Length of 3rd parm value  */
PARMLIST -> PRMPTR   = ADDR(P3VAL);    /* Address of 3rd parm value */

/*********************************************************************/
/* Allocate and initialize fourth parameter block                    */
/*********************************************************************/
ALLOCATE DSIPPRM SET(PARMLIST ->PRMLINK);
PARMLIST = PARMLIST -> PRMLINK;        /* Link to 4th parm in list  */
PARMLIST -> PRMINDEX = 2;              /* 2nd line of multi-parm    */
PARMLIST -> PRMNAML  = LENGTH(P4NAM);  /* Length of 4th parm name   */
PARMLIST -> PRMNAME  = P4NAM;          /* 4th parm name             */
PARMLIST -> PRMTYPE  = PRM_TYPEEBDC;   /* 4th parm is in EBCDIC     */
PARMLIST -> PRMLENG  = LENGTH(P4VAL);  /* Length of 4th parm value  */
PARMLIST -> PRMPTR   = ADDR(P4VAL);    /* Address of 4th parm value */
PARMLIST -> PRMLINK = NULL;            /* Terminate linked list     */

/*********************************************************************/
/* Prepare for REPLY                                                 */
/*********************************************************************/
CALL CNMCMD(HLBPTR,'TRAP AND SUPPRESS ONLY MESSAGES DWO548I');

/*********************************************************************/
/* Send transaction                                                  */
/*********************************************************************/
CALL CNMSNDT(HLBPTR,TRANSID,TRTYPE,DESTNET,DESTDOM,
             DESTTASK,VOCAB,CORR,STARTPTR);
IF HLBRC ¬= CNM_GOOD THEN
  DO;                              /* Unsuccessful CNMSNDT          */
    CALL CNMSMSG(HLBPTR,'CNMSNDT RETURN CODE BAD, HLBRC ='||HLBRC,
                 'MSG','OPER',');
    CALL EXIT(99);
  END;
```

Figure 18. Example of a PL/I Program Using the NetView Bridge Requester API (Part 3 of 5)

```
/*********************************************************************/
/* WAIT for DWO548I message                                          */
/*********************************************************************/
CALL CNMCMD(HLBPTR,'WAIT 60 SECONDS FOR MESSAGES');
IF HLBRC ¬= CNM_MSG_ON_WAIT THEN
  DO;
    CALL CNMSMSG(HLBPTR,'MESSAGE WAIT NOT SATISFIED, HLBRC ='||HLBRC,
                 'MSG','OPER',');
    CALL EXIT(99);
  END;

/*********************************************************************/
/* Process REPLY and retrieve header parameter                       */
/*********************************************************************/
CALL CNMGETP(HLBPTR,BYHEADER,'CORR    ',GPNAMEP,GPINDEX,
             GPDATA,GPDATLEN,GPTYPE,GPQUEUE,GPCURSOR);
IF HLBRC = CNM_GOOD THEN
  DO;
    IF CORR ¬= 'CORRVAL' THEN
      CALL CNMSMSG(HLBPTR,'CORRELATOR NOT MATCHED','MSG','OPER',');
    CALL CNMSMSG(HLBPTR,'CORR = '||GPDATA,'MSG','OPER',');
  END;
ELSE
  CALL CNMSMSG(HLBPTR,'CNMGETP FOR CORR FAILED,HLBRC='||HLBRC,
               'MSG','OPER',');

/*********************************************************************/
/* Retrieve parameter by name                                        */
/*********************************************************************/
GPINDEX = NOT_AN_INDEX;
CALL CNMGETP(HLBPTR,BYNAME,'        ','REASONCODE',GPINDEX,
             GPDATA,GPDATLEN,GPTYPE,GPQUEUE,GPCURSOR);
IF HLBRC ¬= CNM_GOOD THEN
  CALL CNMSMSG(HLBPTR,'CNMGETP FOR REASONCODE FAILED,HLBRC='||HLBRC,
               'MSG','OPER',');

CALL CNMSMSG(HLBPTR,'REASONCODE = '||GPDATA,'MSG','OPER',');
```

Figure 18. Example of a PL/I Program Using the NetView Bridge Requester API (Part 4 of 5)

```
/*********************************************************************/
/* Retrieve parameters by next procedure                             */
/*********************************************************************/
GPCURSOR = 0;
CALL CNMGETP(HLBPTR,BYNEXT,GPNAMEH,GPNAMEP,GPINDEX,
             GPDATA,GPDATLEN,GPTYPE,GPQUEUE,GPCURSOR);

DO WHILE (HLBRC = CNM_GOOD);
  CALL CNMSMSG(HLBPTR,GPNAMEP||' = '||GPDATA,'MSG','OPER',');
  CALL CNMGETP(HLBPTR,BYNEXT,GPNAMEH,GPNAMEP,GPINDEX,GPDATA,GPDATLEN,
               GPTYPE,GPQUEUE,GPCURSOR);
END;

IF HLBRC ¬= CNM_NO_MORE_DATA THEN
  CALL CNMSMSG(HLBPTR,'CNMGETP FAILED,HLBRC = '||HLBRC,
               'MSG','OPER',');

/*********************************************************************/
/* EXIT procedure                                                    */
/*********************************************************************/
EXIT: PROCEDURE(CODE);             /* Exit program                   */

DCL CODE CHAR(2);
HLBRC =CNM_GOOD;
STOP;
END EXIT;

END PLISAMP;
```

Figure 18. Example of a PL/I Program Using the NetView Bridge Requester API (Part 5 of 5)

Example of a Command Processor Using C

Figure 19 on page 71 is an example of a C program that uses the NetView Bridge Requester API. You might want to use this example as a model for creating your own command processor.

Note: VM and VSE do not support C or other HLLs.

```
/*********************************************************************/
/*                                                                   */
/* Sample C language transaction command processor.                  */
/*                                                                   */
/*  This is a sample command processor that sends a transaction      */
/*  across the NetView bridge product.  The TRANSID and all          */
/*  parameters are hard coded and are not related to any specific    */
/*  server or database.  In a real environment, these values should  */
/*  be pulled from some external source.                              */
/*  If the parameters were valid for some data base, then the        */
/*  expected output would be similar to the following :              */
/*                                                                   */
/*    CORR     is CORRVAL                    <-- Cnmgetp method H   */
/*    REASONCODE is        0                 <-- Cnmgetp method P   */
/*    MESSAGES   is RECORD CREATED.          <-- Cnmgetp method N   */
/*    REASONCODE is        0                 <-- Cnmgetp method N   */
/*    Sample C transaction program complete.                         */
/*                                                                   */
/*********************************************************************/
#pragma runopts(NOEXECOPS,NOSTAE,NOSPIE,ISASIZE(4K),ISAINC(4K))

#define NOT_AN_INDEX -32767

/*********************************************************************/
/* Standard C include files                                          */
/*********************************************************************/
#include <string.h>                  /* String Functions          */
#include <stdio.h>                   /* I/O Functions             */
#include <stdefs.h>                  /* Standard Definitions      */
#include <stdlib.h>                  /* Standard Library          */
#include <stdarg.h>                  /* Standard Args             */

#include "dsic.h"                    /* NetView HLL include file  */

/*********************************************************************/
/* 32 byte varying length character string                           */
/*********************************************************************/
typedef struct {
  short size;
  char buffer??(32??);
}D32varc;
```

Figure 19. Example of a C Program Using the NetView Bridge Requester API (Part 1 of 7)

```
/*********************************************************************/
/* External data definitions                                         */
/*********************************************************************/
Dsihlb   *Hlbptr;                    /* Pointer to the HLB         */
Dsivarch *Cmdbuf;                    /* Pointer to Command Buffer  */
Dsiorig  *Origblck;                  /* Pointer to the Origin Block*/

main(int argc, char *argv??(??))
{
  /*******************************************************************/
  /* Internal data definitions                                       */
  /*******************************************************************/

  /*******************************************************************/
  /* Parameter list definitions                                      */
  /*******************************************************************/
  Dsicprm  *parms;                   /* Pointer to parm list       */
  Dsicprm  parm1,parm2,parm3,parm4;  /* Parameter list             */
  Dsivarch p1nam,p2nam,p3nam,p4nam;  /* Parameter names            */
  Dsivarch p1val,p2val,p3val,p4val;  /* Parameter values           */

  /*******************************************************************/
  /* Cnmgetp definitions                                             */
  /*******************************************************************/
  char     gpmethod;                 /* GET method (H, P, N)       */
  char     gpnameh??(9??);           /* Header name area           */
  D32varc  gpnamep;                  /* Parameter name area        */
  int      gpindex;                  /* Index Value                */
  Dsivarch gpdata;                   /* Data buffer area           */
  int      gpdatlen;                 /* Maximum Data length area   */
  short    gptype;                   /* EBCDIC or Binary           */
  int      gpqueue;
  int      gpcursor;                 /* Cursor location in message */

  /*******************************************************************/
  /* Cnmsndt definitions                                             */
  /*******************************************************************/
  char transid??(9??)  = "CREATE  ";  /* Transaction name          */
  char trtype??(5??)   = "QALL";      /* Transaction type          */
  char destnet??(9??)  = "NETA    ";  /* Network destination       */
  char destdom??(9??)  = "CNM01   ";  /* Domain destination        */
  char desttask??(9??) = "BRDGDSP1";  /* Bridge dispatcher name    */
  char vocab??(9??)    = "TABLE1  ";  /* VOCAB table name          */
  char corr??(9??)     = "CORRVAL ";  /* Transaction correlator    */
```

Figure 19. Example of a C Program Using the NetView Bridge Requester API (Part 2 of 7)

```
/*****************************************************************/
/* User definitions                                              */
/*****************************************************************/
int rchlb;
Dsivarch message,command;

/*****************************************************************/
/* Convert parameter pointers from character to hex addresses    */
/*****************************************************************/
sscanf(argv??(1??),"%x",&Hlbptr);
sscanf(argv??(2??),"%x",&Cmdbuf);
sscanf(argv??(3??),"%x",&Origblck);

/*****************************************************************/
/* Initialize Parameter names and values                         */
/*****************************************************************/
Cnmvlc(&1nam,0,"PARMNAME1");    /* 1st parameter name         */
Cnmvlc(&p1val,0,"VALUE1");      /* 1st parameter value        */
Cnmvlc(&p2nam,0,"PARMNAME2");   /* 2nd parameter name         */
Cnmvlc(&p2val,0,"VALUE2");      /* 2nd parameter value        */
Cnmvlc(&p3nam,0,"MULTPARM");    /* Multi-parameter name       */
Cnmvlc(&p3val,0,"MULTVALUE1");  /* 1st multi-parm value       */
Cnmvlc(&p4nam,0,"MULTPARM");    /* Multi-parameter name       */
Cnmvlc(&p4val,0,"MULTVALUE2");  /* 2nd multi-parm value       */

/*****************************************************************/
/* Set up parameter list                                         */
/*****************************************************************/
parms = &parm1;                  /* Start of parameter list   */

/*****************************************************************/
/* Parameter block 1                                             */
/*****************************************************************/
parm1.prmlink  = &parm2;        /* Link to the next parameter */
strcpy(parm1.prmname,p1nam.buffer); /* Parameter name             */
parm1.prmnaml  = p1nam.size;        /* Length of parameter name   */
parm1.prmindex = NOT_AN_INDEX;      /* Index value set to -32767  */
parm1.prmtype  = prm_ebdc;          /* prm_ebdc or prm_int type   */
parm1.prmleng  = p1val.size;        /* Length of parameter value  */
parm1.prmptr   = &p1val.buffer;     /* Address of parameter value */
```

Figure 19. Example of a C Program Using the NetView Bridge Requester API (Part 3 of 7)

```
/*****************************************************************/
/* Parameter block 2                                             */
/*****************************************************************/
parm2.prmlink  = &parm3;        /* Link to the next parameter */
strcpy(parm2.prmname,p2nam.buffer); /* Parameter name            */
parm2.prmnaml  = p2nam.size;        /* Length of parameter name   */
parm2.prmindex = NOT_AN_INDEX;      /* Index value set to -32767  */
parm2.prmtype  = prm_ebdc;          /* prm_ebdc or prm_int type   */
parm2.prmleng  = p2val.size;        /* Length of parameter value  */
parm2.prmptr   = &p2val.buffer; /* Address of parameter value */

/*****************************************************************/
/* Parameter block 3                                             */
/*****************************************************************/
parm3.prmlink  = &parm4;        /* Link to the next parameter */
strcpy(parm3.prmname,p3nam.buffer); /* Parameter name            */
parm3.prmnaml  = p3nam.size;        /* Length of parameter name   */
parm3.prmindex = 1;                 /* Index value set to 1       */
parm3.prmtype  = prm_ebdc;          /* prm_ebdc or prm_int type   */
parm3.prmleng  = p3val.size;        /* Length of parameter value  */
parm3.prmptr   = &p3val.buffer; /* Address of parameter value */

/*****************************************************************/
/* Parameter block 4                                             */
/*****************************************************************/
parm4.prmlink  = NULL;              /* End list with NULL link    */
strcpy(parm4.prmname,p4nam.buffer); /* Parameter name            */
parm4.prmnaml  = p4nam.size;        /* Length of parameter name   */
parm4.prmindex = 2;                 /* Index value set to 2       */
parm4.prmtype  = prm_ebdc;          /* prm_ebdc or prm_int type   */
parm4.prmleng  = p4val.size;        /* Length of parameter value  */
parm4.prmptr   = &p4val.buffer; /* Address of parameter value */

/*****************************************************************/
/* Prepare for reply                                             */
/*****************************************************************/
Cnmvlc(&command,0,"TRAP AND SUPPRESS ONLY MESSAGES DWO548I");
Cnmcmd(&command);
rchlb = Hlbptr->Hlbrc;
if (rchlb != CNM_GOOD)
  Exit_Rtn(1,rchlb);
```

Figure 19. Example of a C Program Using the NetView Bridge Requester API (Part 4 of 7)

```
/*****************************************************************/
/* Send transaction                                              */
/*****************************************************************/
Cnmsndt(transid,trtype,destnet,destdom,desttask,vocab,corr,&parms);
rchlb = Hlbptr->Hlbrc;
if (rchlb != CNM_GOOD)
  Exit_Rtn(2,rchlb);

/*****************************************************************/
/* Wait for messages                                             */
/*****************************************************************/
Cnmvlc(&command,0,"WAIT 60 SECONDS FOR MESSAGES");
Cnmcmd(&command);
rchlb = Hlbptr->Hlbrc;
if (rchlb != CNM_MSG_ON_WAIT)
  Exit_Rtn(3,rchlb);

/*****************************************************************/
/* Get CORR using Cnmgetp with gpmethod of 'H'eader              */
/*****************************************************************/
gpindex  = 0;                       /* Initialize gpindex to 0    */
gpcursor = 0;                       /* Initialize gpcursor to 0   */
gpdatlen = 8;                       /* Set maximum data length    */
gptype   = prm_ebdc;                /* Initialize gptype to EBCDIC*/
gpqueue  = TRAPQ;                   /* Set gpqueue to 1 (TRAPQ)   */
gpmethod = BYHEADER;                /* Set gpmethod to H(BYHEADER)*/
strcpy(gpnameh,"CORR    ");         /* Copy CORR into gpnameh     */
Cnmvlc(&gpnamep,0," ");         /* Blank out gpnamep value    */

Cnmgetp(&gpmethod,gpnameh,&gpnamep,&gpindex,&gpdata,
        gpdatlen,&gptype,gpqueue,&gpcursor);
rchlb = Hlbptr->Hlbrc;
gpdata.buffer??(gpdata.size??) = '\0';
if (rchlb != CNM_GOOD)
 {
  Cnmvlc(&message,0,"Cnmgetp; method H failed.  RC = %d",rchlb);
  Cnmsmsg(&message,MSG,OPER,NULLCHAR);
     }
else
 {
  Cnmvlc(&message,0,"%s is %s",gpnameh,gpdata.buffer);
  Cnmsmsg(&message,MSG,OPER,NULLCHAR);
 }
```

Figure 19. Example of a C Program Using the NetView Bridge Requester API (Part 5 of 7)

```
/*********************************************************************/
/* Get REASONCODE using Cnmgetp with gpmethod of 'P'arameters     */
/*********************************************************************/
gpindex  = NOT_AN_INDEX;            /* Initialize gpindex (-32767)*/
gpcursor = 0;                       /* Initialize gpcursor to 0   */
gpdatlen = 8;                       /* Set maximum data length    */
gptype   = prm_ebdc;                /* Initialize gptype to EBCDIC*/
gpqueue  = TRAPQ;                   /* Set gpqueue to 1 (TRAPQ)   */
gpmethod = BYNAME;                  /* Set gpmethod to P (BYNAME) */
strcpy(gpnameh,"        ");         /* Blank out gpnameh          */
Cnmvlc(&gpnamep,0,"REASONCODE");/* Set gpnamep to parm name   */
Cnmgetp(&gpmethod,gpnameh,&gpnamep,&gpindex,&gpdata,
        gpdatlen,&gptype,gpqueue,&gpcursor);
rchlb = Hlbptr->Hlbrc;
gpnamep.buffer??(gpnamep.size??) = '\0';
gpdata.buffer??(gpdata.size??) = '\0';

if (rchlb != CNM_GOOD)
 {
  Cnmvlc(&message,0,"Cnmgetp method P failed.  RC = %d",rchlb);
  Cnmsmsg(&message,MSG,OPER,NULLCHAR);
 }
else
 {
  Cnmvlc(&message,0,"%s is %s",gpnamep.buffer,gpdata.buffer);
  Cnmsmsg(&message,MSG,OPER,NULLCHAR);
 }

/*********************************************************************/
/* Get remaining parameters using Cnmgetp with gpmethod of 'N'ext */
/*********************************************************************/
gpindex  = NOT_AN_INDEX;            /* Initialize gpindex (-32767)*/
gpcursor = 0;                       /* Initialize gpcursor to 0   */
gpdatlen = 256;                     /* Set maximum data length    */
gptype   = prm_ebdc;                /* Initialize gptype to EBCDIC*/
gpqueue  = TRAPQ;                   /* Set gpqueue to 1 (TRAPQ)   */
gpmethod = BYNEXT;                  /* Set gpmethod to N (BYNEXT) */
strcpy(gpnameh,"        ");         /* Blank out gpnameh          */
Cnmvlc(&gpnamep,0," ");         /* Blank out gpnamep          */
Cnmgetp(&gpmethod,gpnameh,&gpnamep,&gpindex,&gpdata,
        gpdatlen,&gptype,gpqueue,&gpcursor);
rchlb = Hlbptr->Hlbrc;
gpnamep.buffer??(gpnamep.size??) = '\0';
gpdata.buffer??(gpdata.size??) = '\0';
```

Figure 19. Example of a C Program Using the NetView Bridge Requester API (Part 6 of 7)

```
  while (rchlb == CNM_GOOD)
   {
    Cnmvlc(&message,0,"%s is %s",gpnamep.buffer,gpdata.buffer);
    Cnmsmsg(&message,MSG,OPER,NULLCHAR);
    Cnmgetp(&gpmethod,gpnameh,&gpnamep,&gpindex,&gpdata,
            gpdatlen,&gptype,gpqueue,&gpcursor);
    rchlb = Hlbptr->Hlbrc;
   }

  if (rchlb != CNM_NO_MORE_DATA)
   {
    Cnmvlc(&message,0,"Cnmgetp for method N failed. RC = %d",rchlb);
    Cnmsmsg(&message,MSG,OPER,NULLCHAR);
   }

  Exit_Rtn(0,0);
}

/*********************************************************************/
/* Exit routine                                                      */
/*********************************************************************/
Exit_Rtn(int exitcode,int rc)
{
  extern Dsihlb  *Hlbptr;
  Dsivarch  message;

  switch(exitcode)
   {
    case 1 : Cnmvlc(&message,0,"Trap command failed.  RC = %d",rc);
             Cnmsmsg(&message,MSG,OPER,NULLCHAR);
             break;
    case 2 : Cnmvlc(&message,0,"Cnmsndt failed.  RC = %d",rc);
             Cnmsmsg(&message,MSG,OPER,NULLCHAR);
             break;
    case 3 : Cnmvlc(&message,0,"Wait ended abnormally.  RC = %d",
             rc);
             Cnmsmsg(&message,MSG,OPER,NULLCHAR);;              break;
    default: Cnmvlc(&message,0,"Processing complete.");
             Cnmsmsg(&message,MSG,OPER,NULLCHAR);
   }

  Cnmvlc(&message,0,"Sample C transaction program complete.");
  Cnmsmsg(&message,MSG,OPER,NULLCHAR);
  exit(0);
}
```

Figure 19. Example of a C Program Using the NetView Bridge Requester API (Part 7 of 7)

Chapter 5. Starting NetView Bridge, a Database Server, and NetView Bridge Remote Access

This chapter describes the step-by-step process for bringing up NetView Bridge, a database server, and the NetView Bridge remote dispatcher and discusses some methods for automating this process.

The NetView subsystem address space contains the program-to-program interface queues used by NetView Bridge. You must recycle the NetView Bridge dispatcher autotask and the database server every time you assign a different NetView subsystem address space to your NetView environment.

Note: Do not bypass any of the steps in this section. Ensure that you complete each task within each step (in order) before proceeding to the next step.

Step 1. Starting NetView Bridge

To start NetView Bridge, you must first start the NetView Bridge dispatcher by issuing the following command:

```
AUTOTASK OPID=BRIGOPER
```

Where BRIGOPER is the operator ID that you assigned in “Step 1. Adding an Operator ID for the Dispatcher Autotask” on page 8.

This command starts BRIGOPER as a NetView autotask and automatically executes the RTRINIT command as specified in the profile you created in “Step 2. Creating a NetView Bridge Dispatcher Profile” on page 8.

When initialization is complete, you receive message DWO531I, stating that the NetView Bridge initialization is complete.

Step 2. Starting the Database Servers

After you receive message DWO531I, you can start each of your database server copies in accordance with your database server documentation. Consult the documentation for your database server.

If copies of the database server are started before initialization of the autotask has completed, unpredictable results can occur.

If you have followed the procedures outlined in this book and your database server documentation, NetView Bridge is now operational.

Step 3. Starting the NetView Bridge Remote Dispatcher

Before you can start the remote dispatcher, you must initialize the high performance transport task, DSIHPDST. Refer to *Tivoli NetView for OS/390 Installation: Configuring Additional Components* for information on defining the high performance transport. When initialization is complete, you receive message DSI633I.

To access NetView Bridge remotely, you must first start the NetView Bridge remote dispatcher by issuing the following command on both the resident and remote NetView systems:

```
AUTOTASK OPID=REMOPER
```

Where REMOPER is the operator ID that you assigned in “Step 1. Adding an Operator ID for the Remote Dispatcher Autotask” on page 11.

This command starts REMOPER as a NetView autotask and automatically executes the REMOTEBR command as specified in the profile you created in “Step 2. Creating a NetView Bridge Remote Dispatcher Profile” on page 11.

Automating the Startup of NetView Bridge and NetView Bridge Remote Access

You can customize your CNMSTYLE membe so that the AUTOTASK commands described in “Step 1. Starting NetView Bridge” on page 79 and “Step 3. Starting the NetView Bridge Remote Dispatcher” on page 79 execute automatically when the NetView program is started.

Automating the Startup of a Database Server

If your database server is started by an MVS operator command, you can customize your NetView program so that the command to start your database server is automatically issued when the NetView Bridge dispatcher has finished initializing. One method of doing this is to add a NetView automation table entry that traps NetView Bridge message DWO531I and invokes the NetView MVS system operator command from the NetView program.

For information about the NetView automation table, refer to “The Automation Table” in *Tivoli NetView for OS/390 Automation Guide*. For information on issuing the NetView MVS command, refer to the online help.

To successfully automate the startup of a database server, you must ensure that the correct NetView automation table is active, and that the NetView MVS command environment is active.

If you are using MVS EMCS consoles, two requirements must be met before a NetView MVS command executes:

- Ensure that the task is allowed to obtain a console. You do not need to make an entry in CONSOL*xx*, but verify that the console name is not already in use.
- Make sure that the extended console is not blocked by Security Access Facility/ Resource Access Control Facility (SAF/RACF).

If you are using subsystem consoles, several requirements must be met before a NetView MVS command executes:

- The NetView subsystem address space must be started from MVS. This is usually done before starting the primary NetView address space. You receive message CNM563I if the NetView subsystem address space is not active when you issue a NetView MVS command.
- The NetView subsystem address space router task CNMCSSIR must be started. The NetView default is to start his task. Refer to sample CNMSTYLE for more information.

  Message DSI530I indicates that task CNMCSSIR is started and ready.
- If you attempt to start CNMCSSIR when the NetView subsystem address space is not started, message CNM5631 is issued and CNMCSSIR waits until the NetView subsystem becomes active. If you attempt to issue a NetView MVS command when CNMCSSIR is not active, message CNM5601 is issued.

- The NetView MVS command assigns an MVS subsystem console ID to each issuing task. Ask the system programmer to verify that a sufficient number of subsystem consoles has been defined to MVS. For each additional subsystem console that is to be defined, have your system programmer add an entry in SYS1.PARMLIB (CONSOL*xx*) that is similar to the following:

  ```
  CONSOLE DEVNUM(SUBSYSTEM),AUTH(ALL)
  ```

  Reinitialize your MVS system before the additional console definitions become effective.

Chapter 6. Adjusting NetView Bridge to Optimize Performance

This chapter describes some considerations for optimizing the performance of NetView Bridge in your system. Many performance issues are unique to the configuration of your host system. However, certain usage conventions can help you make the best use of NetView Bridge, regardless of your system configuration.

NetView Bridge is an automation tool, but is not intended to support vast quantities of data transport or migration. For this reason, you might want to decrease the amount of data in a given transaction, and the number of transactions in general.

The following sections:

- Describe some of the methods for reducing NetView Bridge traffic.
- Give examples of both desirable and undesirable traffic.
- Describe methods of handling large amounts of traffic when it becomes necessary to do so.

Limiting Traffic Across the Bridge

You can improve the performance of NetView Bridge by using discretion in choosing what to send across the bridge. One method for limiting traffic is to use NetView automation table command list processing to reduce traffic. For detailed information on how to use the automation table, see “Autotasks” in *Tivoli NetView for OS/390 Automation Guide*.

Desirable traffic can be characterized as follows:

- A single transaction that represents multiple units of data. One message stating that several transactions were completed is preferable to individual messages for each occurrence.
- Carefully controlled volumes of traffic. You might find that simply giving some forethought to what you allow across the bridge significantly reduces the amount of normal traffic. Reduce traffic by selectively querying the database rather than using database dumps.

Undesirable traffic can be characterized as follows:

- Large amounts of data that might include:
  - A high volume of repetitious data
  - Numerous single transactions that approach 31K in size
  - Transportation of unimportant data
- Any combination of high volume **plus** large, single transactions
- Any single transaction that results in long (approaching 31K) replies.

Using Multiple Copies of Database Servers to Increase Throughput

If you have made every attempt to reduce unnecessary, or undesirable, traffic, and the demands on a NetView Bridge are increasing, you can increase throughput by adding copies of database servers to a NetView Bridge server set.

Consult the documentation for your database server for instructions on how to create more copies of a server.

Note: Adding too many servers can, at some point, begin to degrade NetView Bridge performance, because large numbers of servers begin to compete for

access to the database. If you experience such degradation of performance, you should experiment with your number of servers to achieve optimum performance in your environment.

Adjusting Queue Limits

When a backlog of transactions occurs, transaction requests are held in the HOLD queue of the program-to-program interface. The NetView Bridge RTRINIT command contains the HQUEUEL operand that sets the size of this queue.

Transactions can back-up in the hold queue. If you are experiencing a problem of this nature, or if you are concerned about your storage use, you can use the DISBQL command to display the number of buffers on the hold queue. You can also adjust the HQUEUEL value of the RTRINIT command to a size that better suits your performance needs. You can use the SETBQL command to reset the size of the queues. See NetView online help for more information on the DISBQL and the SETBQL command.

If any queue limits are reached, message DWO550I is issued, and all subsequent transaction requests received while this condition exists are terminated.

Saving Storage

If the NetView Bridge dispatcher or remote dispatcher is infrequently used, consider changing RES=Y to RES=N on all command model statements to save storage. Refer to “Coding RES=N on Command Model Statements” in the *Tivoli NetView for OS/390 Tuning Guide* for more information on coding RES=N to save storage.

Chapter 7. Reporting a Problem to Tivoli®

This chapter explains how to describe a problem to Tivoli Customer Support so that a resolution can be found and returned to you. For more information on reporting a problem, refer to “Documenting and Reporting Problems” in the *Tivoli NetView for OS/390 Diagnosis Guide*.

Collecting Information

Gather the following information for all problems, including those for which you cannot identify the problem type:

1. Record any abend or return codes, as well as the name of any files involved in the failure.
2. Research the scenario leading to the failure. Answer the following questions:
   - What was the first indication of the problem?
   - What were you trying to do?
   - What did you expect to happen?
   - What actually happened?
   - Has the function worked before?
   - Can you re-create the problem?
3. Note any unique information about the problem or your system, such as:
   - Were other applications running when the problem occurred?
   - Did you modify any of the programs that were running?
4. Record the operating system, version number, release number, and maintenance level. Note any program temporary fixes (PTFs) or authorized program analysis reports (APARs) that have been applied to the operating system or the NetView program.
5. Print a copy of the NetView trace log.

   Note: The NetView trace log is of use only if NetView command list language or REXX code was executing at the time of the failure.
6. Print a copy of the NetView HLL remote interactive debugger (RID) and first failure data capture (FFDC) trace logs. These tools are supported by NetView Bridge service routines CNMSNDT and CNMGETP. For more information, refer to the *Tivoli NetView for OS/390 Customization: Using PL/I and C* and *Tivoli NetView for OS/390 Diagnosis Guide*.

   Note: The NetView Bridge server support API service routines maintain an eight-entry, continuously wrapping trace area. This 48-byte area is referred to as the first failure data capture area (FFDCA). Its name is HLBFFDCA and it is located in the DSIBPHLB control block (for PL/I), and the DSIBCHLB control block (for C).

   You can print the contents of this trace area during job execution by including the appropriate PL/I or C print statements in your service routines. If a failure occurs, this area identifies the server support API module that was running at the time of failure.
7. Print a copy of the system log. The system log is a data set or file in which job-related information, operational data, descriptions of unusual activity, commands, and messages can be stored.
8. Provide a copy of any command list, REXX routine, or HLL service routine (if applicable) that was executing at the time of the problem.

The Tivoli Customer Support representative helps you gather additional information, if needed.

Calling Tivoli Customer Support

Tivoli Customer Support is the first point of contact for NetView Bridge customers who need help with a program problem after installation is complete. When you call Customer Support, a dispatcher asks for customer identification information, such as your account name, access code, and program license number. The dispatcher determines the type of help that you need, assigns a problem number, and places your call on a queue for a Customer Support representative.

Searching the Software Support Database for a Possible Solution

The Tivoli Customer Support representative asks you to describe your problem. The representative uses this information to search RETAIN®, a Tivoli database containing symptoms and resolutions, for known problems and information on problems currently under investigation. The representative may ask for additional information that might help in the search.

If a similar problem description is found, then a solution is probably available. If you have the Information/Access Tivoli licensed program, or IBMLINK, you can search the RETAIN database for similar problems.

If the search does not produce a solution, the representative makes sure that you have the necessary information to discuss the problem with a systems specialist. Your call is then placed on a queue for a systems specialist.

Opening an APAR with Tivoli Customer Support

If a solution cannot be found and if the problem appears to be a new one, the representative may enter an authorized program analysis report (APAR) into the Tivoli Customer Support database.

The systems specialist assigns a number to your APAR. If you are asked to send documentation about your problem to Tivoli, write this APAR number in the upper right corner of each piece of documentation you send.

Both the APAR and other helpful documentation allow the systems specialist to examine the problem in greater detail to develop a solution. If this solution is a coding change, it is put into a program temporary fix (PTF) and sent to you. All information about the solution is entered into the RETAIN database. This procedure keeps the database current with problem descriptions and solutions and makes the information available for future RETAIN searches.

Appendix A. NetView Bridge Initialization Commands

This appendix documents general-use programming interface and associated guidance information.

This appendix describes the NetView Bridge initialization command (RTRINIT) and the NetView Bridge remote access initialization command (REMOTEBR).

RTRINIT Command

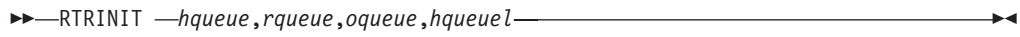
NetView Bridge uses the RTRINIT command to activate the interface between a NetView autotask and a specific set of database servers. This command must be executed under an autotask. See “Step 1. Adding an Operator ID for the Dispatcher Autotask” on page 8 for instructions on creating this autotask.

The RTRINIT command is driven by the profile of a NetView autotask. This autotask serves as the interface between a NetView autotask and a specific set of database servers, and defines three user-provided queue names to the program-to-program interface. These queues are the output queue, to which the database servers send all their output, the ready queue, which contains ready tokens indicating which database servers are available, and the hold queue, which holds requests when no database servers are available. A bridge dispatcher executes the NetView Bridge command RTRINIT to establish the NetView Bridge environment. For more information on the PPI, refer to“Understanding the NetView Program-to-Program Interface” in the *Tivoli NetView for OS/390 Application Programmer’s Guide*.

Note: The operands described in the following syntax are positional and must be placed in the order shown.

The format for the RTRINIT command is:

RTRINIT



Where:

hqueue
Is the name of the hold queue. A PPI queue is created to save transactions that have not been dispatched to a database server. This 1- to 8-character field is required and can contain uppercase alphabetic characters, numeric characters, or $, %, &, @, and #.

rqueue
Is the name of the ready queue. A PPI queue is created to save the READY tokens generated by the database servers. This 1- to 8-character field is required and can contain uppercase alphabetic characters, numeric characters, or $, %, &, @, and #.

oqueue
Is the name of the output queue. A PPI queue is created to receive transaction replies and control messages from the database servers. This queue has the same name as the *tioutq* queue name defined in the CNMETIN service routine

of the server support API. For more information on the service routine CNMETIN, see “Appendix C. Server Support API Commands and Service Routines” on page 107.

This 1- to 8-character field is required and can contain uppercase alphabetic characters, numeric characters, or $, %, &, @, and #.

hqueuel
Defines the limit of the HOLD queue to the PPI. This parameter is required, and must be an integer. Its value must be between 50 and 2000 inclusive. See “Adjusting Queue Limits” on page 84 for more information on the implications of the value you assign to this limit.

Usage Notes

You must execute the RTRINIT command to activate NetView Bridge. If you attempt to route transactions through the bridge without issuing the RTRINIT command, you receive several error messages. Repeat attempts do not produce error messages, however.

If you issue the RTRINIT command from a NetView command list, do not use the ampersand (&) as part of a queue name. The ampersand is defined as a special character in the NetView command list language.

When the RTRINIT command issues an error message, the autotask in which the command is running is still active. You can do one of the following:

- Recycle the autotask as follows:
  1. Issue the following command:
     `EXCMD autotaskname, LOGOFF`

     to log off the autotask from the NetView operator terminal.
  2. Correct the error.
  3. Issue the following command:
     `AUTOTASK OPID=autotaskname`

     to bring up the autotask.
- Correct the error and rerun the command by issuing the following:
  `EXCMD autotaskname, RTRINIT hqueue,rqueue,oqueue,hqueuel`

  from the NetView operator terminal.

Once the RTRINIT command completes successfully, the task may be terminated by severe errors. You can then recycle the autotask as follows:
1. Correct the error.
2. Issue `AUTOTASK OPID=autotaskname`.

The queue limit for *rqueue* and *oqueue* is 2000. If this limit is exceeded, NetView Bridge message DWO550I is issued.

If the queue limit for *hqueuel* (the hold queue) is exceeded, further transactions are discarded, and NetView Bridge message DWO550I is issued. You can increase the limit by changing the value specified by *hqueuel* in the RTRINIT command and issue `AUTOTASK OPID=autotaskname` to bring up the autotask.

In cases where the NetView subsystem address space is down, use the same subsystem interface procedure to bring it up. If you use a different subsystem interface procedure, recycle the autotask.

Return Codes

The return codes for the RTRINIT command are:

Return Code Value	Description
00	The command completed successfully.
20	The command did not complete successfully.

REMOTEBR Command

The REMOTEBR command enables the NetView Bridge remote dispatcher to register as an HP application on the high performance transport API. This command is driven by the profile of the NetView Bridge remote dispatcher autotask. The autotask must be running in both the remote NetView program that is sending transactions and receiving replies, and in the resident NetView program in which the database server resides. Only one autotask in each NetView program can issue this command at a time. Consecutive invocations of this command by other tasks results in message DWO534I being issued.

Note: Although you can execute REMOTEBR from the same autotask NetView Bridge is using to communicate with a local database server, this setup is not recommended. Issue the REMOTEBR command from a new autotask.

See “Step 1. Adding an Operator ID for the Remote Dispatcher Autotask” on page 11 for instructions on creating this autotask.

The syntax for the REMOTEBR command is:

REMOTEBR

Usage Notes

When the REMOTEBR command issues an error message, the autotask in which it is running is still active. You can do one of the following:

- Recycle the autotask as follows:
  1. Issue the following command:
     `EXCMD autotaskname, LOGOFF`

     to log off the autotask from the NetView operator terminal.
  2. Correct the error.
  3. Issue the following command:
     `AUTOTASK OPID=autotaskname`

     to bring up the autotask.
- Correct the error and rerun the command by issuing the following command:
  `EXCMD autotaskname, REMOTEBR`

  from the NetView operator terminal.

Once the REMOTEBR command completes successfully, the task can be terminated by severe errors. You can then recycle the autotask as follows:

1. Correct the error.
2. Issue `AUTOTASK OPID=`*`autotaskname`*.

Return Codes

The return codes for the REMOTEBR command are:

Return Code Value	Description
0	The command completed successfully.
20	The command was not completed successfully.

Appendix B. NetView Bridge Requester API Commands and Service Routines

This appendix documents general-use programming interface and associated guidance information.

This appendix describes REXX commands, NetView command lists, and HLL service routines associated with the NetView Bridge Requester API part of NetView Bridge. The procedures for programming the NetView Bridge Requester API are in “Chapter 4. Writing a NetView Bridge Command Procedure Using the NetView Bridge Requestor API” on page 55. Use the reference information in this appendix as a guide for creating your own NetView Bridge command procedures.

REXX and NetView command list commands are listed (by function), and described in detail, beginning with “TRANSND—Send Transaction Request to Database Server”.

HLL service routines are listed by name and described in detail, beginning with “CNMSNDT (CNMSENDTR)—Send Transaction Request to Database Server” on page 96. Each service routine has an explanation of its use and syntax, as well as complete descriptions of each field and operand.

REXX and NetView Command List NetView Bridge Commands

You can issue two commands, the TRANSND and TRANRCV commands, from the NetView interface using REXX routines or NetView command lists to request transactions and process the results.

This section describes these commands and their operands. Figure 16 on page 58 shows a sample REXX EXEC that uses the TRANSND and TRANRCV commands. Figure 17 on page 62 shows a sample NetView command list that uses the TRANSND and TRANRCV commands. These samples assume that the command procedure is being run under an Operator Station Task (OST). Running under the primary program operator interface task (PPT) is not recommended because you cannot wait for transaction replies.

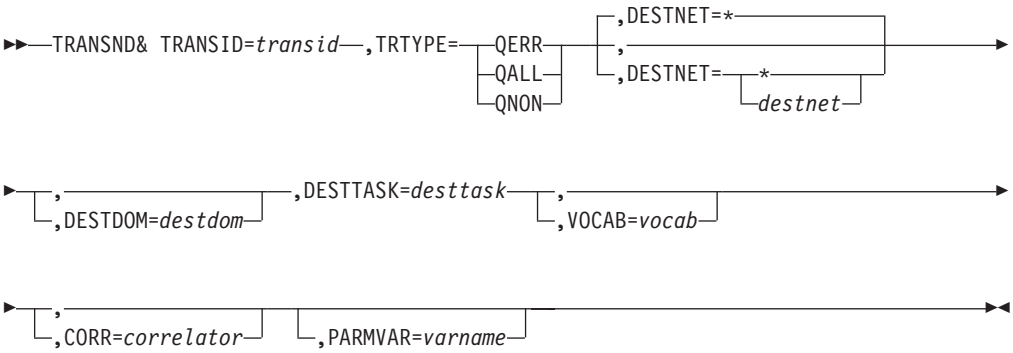
TRANSND—Send Transaction Request to Database Server

The TRANSND command is used to build and send a transaction request to the transaction processor of a database server. The TRANSND command can run only under an operator station task (OST) or a primary POI task (PPT). Running under the PPT is not recommended because you cannot wait for transaction replies. It cannot run under a NetView-NetView task (NNT).

After you send a remote transaction using the high performance transport API, some parameter information is saved. In the event of an error in the send, this information must be retrieved for posting to message DWO546I. Initially information is saved for 2 minutes. However, you can adjust this time by issuing `DEFAULTS NOREPLY = x` (where *x* is the time in seconds). If the queue that holds the information is discarded before the error message is received, UNKNOWN is displayed in the field of the unknown parameter in the DWO546I error message.

The syntax for the TRANSND command is:

TRANSND



Where:

TRANSID=*transid*
Is the name of the transaction to be executed. TRANSID is a required operand that must be 1—8 alphanumeric characters.

TRTYPE={QERR|QALL|QNON}
Indicates the type of transaction to be performed. TRTYPE is a required operand, and it must be entered as uppercase characters. The following are the possible values:

QERR Indicates that this is a request transaction, but only error responses are to be returned to the requesting task.

QALL Indicates that this is a request transaction, but all operational and error response data is to be returned to the requesting task.

QNON Indicates that this is a request transaction, but that no responses (error condition or operational) are to be returned to the requesting task.

DESTNET={*|*destnet***}**
Is an optional 1- to 8-character (uppercase) field that identifies the ID of the network where the bridge dispatcher, server address space, and target database reside. If you do not specify this field, or if you specify DESTNET=*, VTAM determines the network ID based on the LU name of the remote node. If two nodes in two different networks have the same LU name, VTAM will sometimes locate one and sometimes locate the other, depending on the active configuration.

DESTDOM=*destdom*
Is an optional 1- to 8-character (uppercase) field that identifies the ID of the domain where the bridge dispatcher, server address space, and target database reside. If you do not specify this field, it defaults to the local network ID.

DESTTASK=*desttask*
Is a required 1- to 8-character (uppercase) field that identifies the task ID of the bridge dispatcher autotask that manages the server that processes the transaction.

VOCAB=*vocab*
Is an optional operand that specifies the name of the context used to interpret the operand names in the request. The context is defined by the server. (See the documentation for your particular server for details on vocabulary restrictions.) This is a mechanism for converting operand names that are

recognized by NetView to names that can be recognized by the target database. The value of this operand, if specified, is 1–8 alphanumeric characters. If you omit this operand, the default context defined by the target database is used.

CORR=*correlator*
Is an optional operand that can be used to associate a response with any data that was kept by the requestor at the time the request transaction was issued. If you specify this operand, the value must be 1–8 alphanumeric characters. If you do not specify a value, an 8-character alphanumeric value is generated by this command and returned in a variable named CORREL in the format DDHHMMSS, where:

DD Is the day
HH Is the hour
MM Is the minute
SS Is the number of seconds

The invoker can use the value generated in the same manner that it can use a user-supplied value.

PARMVAR=*varname*
Is an optional operand you can use to specify the name of a REXX or NetView command list variable. The value of this variable is a list of names of operands (delimited by blanks) to be sent with the request. The values of the operands listed in PARMVAR are obtained by treating each name specified as the name of a REXX or NetView command list variable. The value of the variable is the value of the operand.

REXX limits the variable name length to 31 characters, and the NetView command list language limits the variable name length to 11 characters. All alphabetic characters in this name must be entered as uppercase characters.

If a variable with the name you specified does not exist, the value of the operand is assumed to be null (length of 0).

If a REXX variable is a stem with an index (for example, A*n* where *n* is numeric), the operand A is assumed to be an array (it has subparameters) with indexes 1 through *n*. The number of indexes (size of the array) must be indicated in A.0. The stem variable must be represented in the PARMVAR in the form of *variable-name* operand (for example, A.). This form is used when handling lines of text in which each line of text is an entry in the array. The NetView command list language does not provide the array-type variable.

NetView command list language variable names are listed in PARMVAR as the name of the variable without the leading ampersand (&). All parameter values are designated to have a value-type indicator of X'00EE' in the generated transaction showing that it is an EBCDIC string.

Usage Notes

The maximum length of the transaction operands and their values that can be specified on the TRANSND command is approximately 31K. A transaction request that is larger than the maximum length is rejected, and NetView Bridge message DWO568I is issued.

Return Codes

The return codes for this command are:

Return Code Value	Description
0	Successful.
4	Processing error encountered—no request sent. Error message was issued.

TRANRCV—Receive a Transaction Reply

The TRANRCV command is issued by a REXX or NetView command list after that command list has trapped message DWO548I. The TRANRCV command contains the operands of a transaction reply.

The syntax for the TRANRCV command is:

TRANRCV

Where:

PARMVAR=*varname*

Is an optional operand that specifies the name of a REXX or NetView command list variable. The value of this variable is a list of names of operands, delimited by blanks, returned with the reply. The parameters are returned from the transaction processor. The parameter values can be obtained by parsing the PARMVAR variable.

The name of the parameter from the transaction becomes the name of the variable as long as it satisfies the restrictions for variable names specified for the language. The value of the parameter becomes the value of the variable.

NetView command list language variable names are stored in the PARMVAR variable without the ampersand.

Variable names that are too long for the language are truncated after the maximum number of characters allowed is reached. REXX limits variable names to 31 characters, and the NetView command list language limits variable names to 11 characters.

If a parameter is actually an array, the data is stored in different variable names. How this data is stored depends on whether REXX or the NetView command list language is used. If REXX is used, variables are named A.*n*, where A is the name of the parameter and *n* is the index into the array. A.0 contains the number of entries in the array. If a NetView command list is used, no more than the first 9 characters of the parameter name are used, followed by the two digits of the index (an index value of 1 is put into the name as 01, and so on).

When a transaction reply returned from a database server is longer than 31K, it is truncated to 31K and sent to the NetView Requester API. The truncation is indicated to the API requester by a CNM_TRUNCATION parameter with the

value YES. The CNM_TRUNCATION parameter is the last parameter returned in the PARMVAR if the truncation occurred in a transaction reply.

The following variable names for the header parameters will always be set by this command:

RPTRANSID
Is a 1- to 8-character field that identifies the received transaction reply.

RPTRTYPE
Is a 4-character field that identifies the type of transaction that is being received.

RPDESTNET
Is a 1- to 8-character field that identifies the destination network ID of the received transaction. This value should be the network ID of the NetView in which this command is running.

RPDESDOM
Is a 1- to 8-character field that identifies the destination domain ID of the received transaction. This value should be the domain ID of the NetView in which this command is running.

RPDESTSK
Is a 1- to 8-character field that identifies the destination task ID of the received transaction. This value should be the NetView task ID in which this command is running.

RPORIGNET
Is a 1- to 8-character field that identifies the originating network ID of the received transaction.

RPORIGDOM
Is a 1- to 8-character field that identifies the domain ID of the received transaction reply.

RPORIGTSK
Is a 1- to 8-character field that identifies the NetView autotask that serves the database server that sent this transaction reply.

RPCORREL
Is a 1- to 8-character correlator field from the transaction reply received.

RPVOCAB
Is a 1- to 8-character field that contains the name of the context used to interpret the parameters in the transaction reply.

RPRESPCODE
Is an 8-character field that contains the response code value that was returned with the transaction reply.

Note: These variables do not appear in the variable identified with the PARMVAR parameter. They are individually set by the TRANRCV command.

Return Codes

The return codes for this command are:

Return Code Value	Description
0	Successful.

Return Code Value	Description
2	Transaction parameter name or value is truncated—processing continues.
4	Processing ended due to failure—error message is issued.

NetView Bridge HLL Service Routines

You can issue two commands, CNMSNDT and CNMGETP, from C and PL/I programs using the NetView Bridge Requester API. These commands invoke service routines and enable you to request transactions and to process the results from NetView.

This section describes these commands and their operands. Figure 18 on page 66 shows a sample PL/I program that uses the CNMSNDT and CNMGETP commands. Figure 19 on page 71 shows a sample C program that uses the CNMSNDT and CNMGETP commands. These samples assume that the command procedure is being run under an Operator Station Task (OST). Running under the primary program operator interface task (PPT) is not recommended because you cannot wait for transaction replies. Also, VM and VSE do not support PL/I or C.

Four types of parameters can be passed to the NetView Bridge Requester API service routines. They are:
- Pointer variables
- Integer variables
- Fixed-length character strings
- Varying-length character strings

The NetView Bridge Requester API service routines require you to declare, initialize, and pass parameters to the API service routines according to PL/I or C programming language specifications. For more information on these specifications, refer to *OS PL/I Version 2 Programming: Language Reference* and “Coding Your PL/I Program: Environment, Interfaces, and Restrictions” or “Coding Your C Program: Interfaces and Restrictions” in *Tivoli NetView for OS/390 Customization: Using PL/I and C*.

CNMSNDT (CNMSENDTR)—Send Transaction Request to Database Server

The CNMSNDT service routine sends a transaction request to the database server.

After you send a remote transaction using the high performance transport API, some parameter information is saved. In the event of an error in the send, this information must be retrieved for posting to message DWO546I. Initially information is saved for two minutes. However, you can adjust this time by issuing `DEFAULTS NOREPLY = x` (where x is the time in seconds). If the queue that holds the information is discarded before the error message is received, UNKNOWN is displayed in the field of the unknown parameter in the DWO546I error message.

The syntax for the CNMSNDT command is:

PL/I CALL FORMAT:

CALL CNMSNDT(*hlbptr,sttransid,sttrtype,stdestnet,stdestdom,stdesttask, stvocab,stcorr,stparms*)

PL/I MACRO FORMAT:

CNMSENDTR TRANSID(*sttransid*) TRTYPE(*sttrtype*) DESTNET(*stdestnet*) DESTDOM(*stdestdom*) DESTTASK(*stdesttask*) VOCAB(*stvocab*) CORR(*stcorr*) PARMS(*stparms*)

C INVOCATION:

void Cnmsndt(char **sttransid*,char **sttrtype*,char **stdestnet*, char **stdestdom*,char **stdesttask*,char **stvocab*, char **stcorr*,Dsicprm ***stparms*)

Where:

hlbptr
Is a 4-byte pointer field containing the address of the HLL control block that was passed to the HLL program (DSIPHLB for PL/I or DSICHLB for C). This control block is described in “Coding Your PL/I Program: Environment, Interfaces, and Restrictions” and “Coding Your C Program: Interfaces and Restrictions” in *Tivoli NetView for OS/390 Customization: Using PL/I and C*.

sttransid
Is a fixed-length 8-character operand that contains the name of the transaction to be executed. If this field contains fewer than 8 characters, it must be left-justified and padded with blanks. No embedded blanks are allowed in this value.

sttrtype
Is a 4-character operand that indicates the type of transaction processing to be performed. The database server uses the value specified in this field to determine what response is to be returned for this request. The following are the possible values:

QERR Indicates that this is a request transaction, but only error responses are to be returned to the requesting task.

QALL Indicates that this is a request transaction, but all operational and error response data is to be returned to the requesting task.

QNON Indicates that this is a request transaction, but no responses (error condition or operational) are to be returned to the requesting task.

stdestnet
Is a fixed-length 8-character field that identifies the network that contains the bridge dispatcher, the server address space, and the target database. If this value is fewer than 8 characters, it must be left-justified and padded with blanks. If the value of the parameter is blanks or *, this field defaults to the network ID determined by VTAM based on the domain name specified in *stdestdom*. If two nodes in two different networks have the same LU name, VTAM will sometimes locate one and sometimes locate the other, depending on the active configuration.

stdestdom
Is a fixed-length 8-character field that identifies the domain that contains the bridge dispatcher, the server address space, and the target database. If this

value is fewer than 8 characters, it must be left-justified and padded with blanks. If the value of the parameter is blanks, this field defaults to the local domain ID.

stdesttask
Is a fixed-length 8-character field that identifies the task ID of the autotask that serves the database server. This autotask processes the transaction. If this value is fewer than 8 characters, it must be left-justified and padded with blanks. No embedded blanks are allowed in this value.

stvocab
Is a fixed-length 8-character field that identifies the context that is used to interpret the parameter names in the transaction. If this field is left blank, the default context is used. The default context is defined in your database server.

stcorr
Is a fixed-length 8-character field that associates a response with any data that was kept by the requestor at the time the transaction was issued. If no correlator is required, set the value of this field to blanks.

stparms
Is a field that contains a pointer to a linked list of parameter blocks. If no parameters are to be sent to a transaction, set this field to either a C null (X'00000000') or a PL/I null (X'FF000000').

Note: NetView provides the mapping of the parameter block in PL/I and C under the include file names DSIPPRM (for PL/I) and DSICPRM (for C). These files are included by DSIPLI and DSIC. Therefore, you need to include DSIPLI or DSIC in your program.

The following is the structure of a parameter block:

prmlink
Contains a pointer to the next parameter block in the chain. If this is the last parameter in the chain, set this field to either a C null (X'00000000') or a PL/I null (X'FF000000').

prmnaml
Contains a halfword binary integer that defines the length of the parameter name. This value must be in the range 1–31.

prmname
Is a fixed-length 31-byte field containing the name of the parameter.

prmindex
Contains a signed halfword binary integer that identifies the index value within a parameter that happens to be an array. A value of minus 32,767 is interpreted to mean that this parameter is not an array. Parameter arrays are intended to handle multiple values for a given parameter (for example, lines of text in a problem description).

prmtype
Contains a halfword binary integer that identifies the type of parameter. This field has two possible values:

X'00EE'
Indicates that the parameter value is an EBCDIC character string.

X'00BB'
Indicates that the parameter value is a binary integer.

prmleng
Contains a halfword binary integer that identifies the length of the parameter value. The value of this field must be less than 31,000.

If the parameter type value (*prmtype*) is X'00EE', indicating that this is an EBCDIC character string, this field contains the length of the character string (in number of bytes). If the parameter type value is X'00BB', indicating that this is a binary integer, this field contains the length of the fixed binary integer variable, which is used to carry the binary integer. For example:

- In PL/I, the length is 4 for a FIXED BIN(31,0) variable, or 2 for a FIXED BIN(15,0) variable.
- In C, the length is 4 for a long integer variable, or 2 for a short integer variable.

Note: The actual maximum length of any parameter value depends on the length of all fields in the transaction. The maximum length of any transaction is 31K. If the transaction request exceeds the maximum length the transaction is rejected. For complete details on calculating the size of a transaction, see “Understanding Transaction Size Limitations” on page 55.

prmptr
Contains a pointer to the value of the parameter. If you do not specify a value, set this field to either a C null (X'00000000') or a PL/I null (X'FF000000').

Return Codes

The return codes for this command are:

Return Code Name	Value	Description
CNM_GOOD	0	Successful.
CNM_BAD_INVOCATION	4	Nonzero return code.
CNM_NO_STORAGE	24	Storage allocation failure.
CNM_NOT_IN_ASYNC	44	Deregistration unsuccessful. Issued from an exit.
CNM_BAD_TIMEOUT	56	Time-out value is not valid.
CNM_BAD_LENGTH	88	MDS_MU length is not valid.
CNM_TASK_INACTIVE	220	DSI6DT task is inactive.
CNM_REQ_TOO_LONG	288	The transaction request generated exceeds 31K.
CNM_BAD_TASK	292	Invalid task type. This service routine can be invoked only under an OST or a PPT.
CNM_BAD_TRAN_HDR	296	One of the transaction header parameters contains an invalid value.
CNM_PROCESSING_ERROR	300	A severe error condition was encountered when the service routine attempted to build the transaction request.
CNM_BAD_DATA_TYPE	400	Data type is not valid.

Return Code Name	Value	Description
CNM_BAD_DATA	404	DATA missing or not valid.
CNM_SAME_APPL	408	MS application cannot send to itself.
CNM_OAPPL_NOT_REG	416	MS application is not registered.
CNM_BAD_SAPPL	420	Operations management served application is not registered.
CNM_BAD_UOW	424	UOW missing or not valid.
CNM_BAD_RTI	428	RTI missing or is invalid.
CNM_BAD_OAN	432	OAN missing or is invalid.
CNM_BAD_DAN	436	DAN missing or is invalid.
CNM_BAD_OAPPL	440	Origin application name is invalid.
CNM_BAD_DNETID	444	Destination network ID missing or is invalid.
CNM_BAD_DLU	448	Destination LU name missing or is invalid.
CNM_BAD_DAPPL	452	Destination application name missing or is invalid.
CNM_BAD_OII	456	OII in RTI does not match TVBOPID.
CNM_BAD_REPLY	460	Reply is invalid.
CNM_BAD_MUTYPE	464	Bad MUTYPE given.
CNM_BAD_SYNCH	468	Bad SYNCH option.
CNM_BUSY	472	User list is full.
CNM_BAD_MQS	1000 + X	MQS failed while sending a transaction request to DESTTASK. X is the return code from DSIMQS.
CNM_BAD_PUSH	4000 + Y	Nonzero return code, Y, from DSIPUSH macro. [1]
CNM_BAD_CES	9000 + Y	Reply command is invalid. Y is the return code from DSICES.
CNM_BAD_TRAN_PARM	22000 + *n*	The *n*th parameter block in the link list pointed to by the *stparms* field contains an invalid *prmnaml*, *prmleng* value, or both.
Note: Receipt of any nonzero return code indicates that the transaction was not sent.		

CNMGETP (CNMGETPARM)—Get Transaction Reply Parameters

The CNMGETP service routine allows you to extract the contents of NetView Bridge message DWO548I during transaction reply processing. It is important that the DWO548I message remain on the queue until CNMGETP is finished extracting the required data.

1. See *Tivoli NetView for OS/390 Customization: Using Assembler* for more information.

The syntax for the CNMGETP command is:

PL/I CALL FORMAT:

CALL CNMGETP(*hlbptr,gpmethod,gpnameh,gpnamep,gpindex,gpdata, gpdatlen,gptype,gpqueue,gpcursor*)

PL/I MACRO FORMAT:

CNMGETPARM METHOD(*gpmethod*) HNAME(*gpnameh*) PNAME(*gpnamep*) INDEX(*gpindex*) DATA(*gpdata*) LENG(*gpdatlen*) TYPE(*gptype*) QUEUE(*gpqueue*) CURSOR(*gpcursor*)

C INVOCATION:

void Cnmgetp(char **gpmethod*,char **gpnameh*,void **gpnamep*, int **gpindex*,void **gpdata*,int *gpdatlen*, short **gptype*,int *gpqueue*,int **gpcursor*)

Where:

hlbptr
Is a 4-byte pointer field containing the address of the HLL control block. This control block is described in "Coding Your PL/I Program: Environment, Interfaces, and Restrictions" and "Coding Your C Program: Interfaces and Restrictions" in *Tivoli NetView for OS/390 Customization: Using PL/I and C*.

gpmethod
Is a 1-character field that identifies the method to use for retrieving parameters from the transaction reply. It must contain one of the following values:

H Retrieve the transaction header parameter and its value.

P Retrieve the value of a transaction parameter by specifying the name of the transaction parameter

N Retrieve the name and value of the next transaction parameter, and start searching from the position pointed to by GPCURSOR.

gpnameh
Is an 8-character fixed-length field used to identify the transaction header parameter that is being requested. It must be left-justified and padded with blanks. This field is required if you specify H in *gpmethod*. It is not required if you specify P or N.

The following names are valid:

TRANSID
Specifies that the name of the transaction reply is returned.

CORR Specifies that the correlator of the transaction reply is returned.

VOCAB
Specifies that the context of parameter names is returned.

TRTYPE
Specifies that a value indicating that this is a transaction reply (RPLY) is returned.

ORIGNET
Specifies that the origin network ID of the transaction reply is returned.

ORIGDOM
Specifies that the origin domain ID of the transaction reply is returned.

ORIGTASK
Specifies that the origin task ID of the transaction reply will be returned. The only task ID that is returned is the ID of the server dispatcher autotask.

DESTNET
Specifies that the destination network ID of the transaction reply is returned.

DESTDOM
Specifies that the destination NetView domain ID of the transaction reply is returned.

DESTTASK
Specifies that the destination task ID of the transaction reply will be returned. The value returned should be the NetView task ID under which this routine is running.

RESPCODE
Specifies that the response code for the transaction reply is returned.

gpnamep
Is a 31-character varying-length field with two different uses:

- When you specify P with *gpmethod*, this field (on input) must contain the name of the parameter whose value is to be retrieved.
- When you specify N with *gpmethod*, this field (on output) is used to receive the name of the retrieved parameter. The search for this parameter begins at the location pointed to by *gpcursor*. The value of *gpcursor* is then set to point to the next parameter.

This field is required if you specify P or N in *gpmethod*. It is not required if you specify H.

gpindex
Contains a signed 4-byte integer that has two different uses:

- When you specify P with *gpmethod*, this field (on input) contains the index value of an array parameter whose value is to be retrieved. You must specify an index value of –32,767 if the parameter is not an array parameter.
- When you specify N with *gpmethod*, this field (on output) is used to receive the index value of the retrieved parameter. The search for this parameter begins at the location pointed to by *gpcursor*. A value of –32,767 is returned if this parameter is not an array type parameter.

This field is required if you specify P or N in *gpmethod*. It is not required if you specify H.

gpdata
Is a varying-length character field containing the value for the parameter being returned.

If the parameter type value (*gptype*) is X'00EE', indicating that this is an EBCDIC character string, this field contains the length of the character string in either the PL/I character varying format or the C Dsivarch format. If the parameter type value is X'00BB', indicating that this is a binary integer, this field contains the binary integer value in the following format:

- The first two bytes indicate the length of the fixed binary integer variable which was used to carry the integer in the transaction reply. For example:
  - In PL/I, the length is 4 for a FIXED BIN(31,0) variable, or 2 for a FIXED BIN(15,0) variable.

– In C, the length is 4 for a long integer variable, or 2 for a short integer variable.

- The remaining two or four bytes contain the binary integer value.

gpdatlen
Is a 4-byte integer that specifies the maximum length of *gpdata*.

If the value specified by *gpdatlen* is less than the length of the parameter value to be returned, the truncated parameter value is returned in *gpdata*, and a return code of CNM_DATA_TRUNC is issued. The full length of the parameter value that was truncated is stored in HLBLENG (Hlbleng).

If the value specified by *gpdatlen* is equal to or greater than the length of the parameter value to be returned, and a return code of CNM_GOOD is issued, the length of the returned parameter value is stored in HLBLENG (Hlbleng).

gptype
Is a 2-byte integer field containing the type of the parameter value returned in *gpdata*. One of the following values is set when this procedure returns:

X'00EE'
Indicates that the returned parameter value is and EBCDIC character string.

X'00BB'
Indicates that the returned parameter value is an integer.

This field is required if you specify P or N in *gpmethod*. It is not required if you specify H.

gpqueue
Is a 4-byte integer field that specifies the queue that holds the DWO548I message containing the transaction reply. For complete details on this queue identifier, see “PL/I High-Level Language Services” and “C High-Level Language Services” in *Tivoli NetView for OS/390 Customization: Using PL/I and C*.

gpcursor
Is a 4-byte integer field used to keep track of the position in the scan through the transaction. If you specify N, this field must be initialized to a binary zero before the first call to CNMGETP. Use of this field with a nonzero value can cause unpredictable results. This field is updated each time the CNMGETP is called.

This field is required if you specify N in *gpmethod*. It is not required if you specify H or P.

Note: When a transaction reply returned from a database server is longer than 31K bytes, it is truncated to 31K and sent to the NetView Requester API. The truncation is indicated to the API requester by returning a CNM_TRUNCATION parameter with the EBCDIC value YES. The CNM_TRUNCATION parameter is the last parameter returned in a transaction reply if the truncation occurred in the reply.

Return Codes

The return codes for this command are:

Return Code Name	Value	Description
CNM_GOOD	0	Successful.

Return Code Name	Value	Description
CNM_NO_STORAGE	24	Storage allocation failure.
CNM_DATA_TRUNC	40	Data is too long for *gpdata* buffer—truncation occurred. Actual length of the returned data is indicated in *hblengh*.
CNM_BAD_QUEUE	72	Queue number specified is out of range.
CNM_QUEUE_EMPTY	80	Specified queue is empty.
CNM_BAD_LENGTH	88	Specified *gpdatlen* is not valid.
CNM_BAD_NAME	108	Either the requested *gpnameh* is not valid (*gpmethod*=H), or the requested *gpnamep* did not exist in the transaction (*gpmethod*=P).
CNM_BAD_OPTION	128	Invalid *gpmethod* is specified.
CNM_BAD_ADDR	160	The *gpdata* data area address is not valid.
CNM_NO_DATA	268	Either the specified *gpnameh* is valid, but no value was found (*gpmethod*=H), or the specified *gpnamep* existed in the transaction but has a value indicating a length of zero (*gpmethod*=P).
CNM_INVALID_MESSAGE	272	First message on the specified queue is not a two-line message with a valid identifier, or the second line of the message is not valid.
CNM_NO_MORE_DATA	280	No more data to be returned (*gpmethod*=N).
CNM_BAD_CURSOR	284	Specified *gpcursor* value is out of range (*gpmethod*=N).
CNM_BAD_TASK	292	Invalid task type. This service routine can be invoked only under an OST or a PPT.

Parameter Usage Reference

Table 5 summarizes the content of fields required for using the *gpmethod* parameter of the CNMGETP service routine.

Table 5. Content Fields for gpmethod Parameter of CNMGETP Service Routine

Parameter Name	Used when *gpmethod*=	Input/Output	Description
gpmethod	P,N,H	Input	Decoding method to use.
gpnameh	H	Input	Name of the header parameter.
gpnamep	P	Input	Name of the requested transaction parameter.
	N	Output	Name of the next transaction parameter.

Table 5. Content Fields for gpmethod Parameter of CNMGETP Service Routine (continued)

Parameter Name	Used when *gpmethod*=	Input/Output	Description
gpindex	P	Input	Index of specified parameter.
	N	Output	Index of the next parameter.
gpdata	P,N,H	Output	Area to receive requested information.
gpdatlen	P,N,H	Input	Length of the provided data area.
gptype	P,N	Output	Type of parameter value.
gpqueue	P,N,H	Input	HLL API queue number.
gpcursor	N	Input/ Output	Position to scan for the next transaction parameter.

CNMSNDT and CNMGETP Debugging Support

The CNMSNDT and CNMGETP service routines support the NetView HLL API remote interactive debugger (RID) and the first failure data capture trace (FFDCT). For more information on these debugging tools, refer to“High-Level Language Services” in *Tivoli NetView for OS/390 Customization: Using PL/I and C*and “Diagnostic Tools for the NetView Program” in *Tivoli NetView for OS/390 Diagnosis Guide*.

Notes:

1. Both of these service routines provide input parameters and their associated values on entry and exit to the RID facility, when it is active.
2. Both of these service routines provide the following module IDs to the NetView FFDCT:
   - 017—CNMGETP function
   - 018—CNMSNDT function

Appendix C. Server Support API Commands and Service Routines

This appendix documents general-use programming interface and associated guidance information.

This appendix contains descriptions of the service routines associated with the NetView Bridge server support API. You can use the reference information in this section as a guide to create your own database server or transaction processor. For detailed information on creating your own database server or transaction processor, see “Chapter 3. Creating a Database Server” on page 15.

The server support API consists of HLL service routines that can be issued by transaction processors. The service routines are listed and described in detail, beginning with “CNMEGTP (CNMETRPARM) Service Routine—Get Transaction Request Parameters”. Each service routine includes an explanation of its use and syntax, as well as a complete descriptions of each field and operand.

Sample service routines (written in PL/I or C) are also provided in this manual. Figure 13 on page 25 shows a sample PL/I service routine. Figure 14 on page 42 shows a sample C service routine. You can use these examples as models for creating your own routines.

PL/I and C Service Routines Issued from the Server Support API

This section describes the high-level language (HLL) service routines that are provided with NetView Bridge. These service routines can only be invoked by programs running outside of the NetView address space.

The server support API service routines require you to declare, initialize, and pass parameters to the API service routines according to PL/I or C programming language specifications. For more information on these specifications, refer to *OS PL/I Version 2 Programming: Language Reference* “Coding Your PL/I Program: Environment, Interfaces, and Restrictions” or “Coding Your C Program: Interfaces and Restrictions” in *Tivoli NetView for OS/390 Customization: Using PL/I and C*, and *SAA: Common Programming Interface C Reference*.

CNMEGTP (CNMETRPARM) Service Routine—Get Transaction Request Parameters

The CNMEGTP service routine enables you to extract the contents of a transaction request after invoking the CNMENTR routine.

The syntax for the CNMEGTP command is:

PL/I CALL FORMAT:

CALL CNMEGTP(*bhlbptr,gpmethod,gpnameh,gpnamep,gpindex,gpdata, gpdatlen,gptype,gpcursor*)

PL/I MACRO FORMAT:

CNMETRPARM METHOD(*gpmethod*) HNAME(*gpnameh*) PNAME(*gpnamep*) INDEX(*gpindex*) DATA(*gpdata*) LENG(*gpdatlen*) TYPE(*gptype*) CURSOR(*gpcursor*)

C INVOCATION:

void Cnmegtp(char **gpmethod*,char **gpnameh*,void **gpnamep*, int **gpindex*,void **gpdata*,int *gpdatlen*, short **gptype*,int **gpcursor*)

Where:

bhlbptr
Is a fullword field containing the anchor pointer used by the server support API.

gpmethod
Is a 1-character field that identifies the method to use for retrieving parameters from the transaction request. This field must contain one of the following values:

H Retrieve transaction header parameter and its value.

P Retrieve value of a transaction parameter by specifying the name of the transaction parameter.

N Retrieve name and value of next transaction parameter, start searching from the position pointed to by *gpcursor*.

gpnameh
Is an 8-character, fixed-length field used to identify the transaction header parameter that is being requested. It must be left-justified and padded with blanks. This field is required if you specify H in *gpmethod*. It is not required if you specify P or N.

The following names are valid:

TRANSID
Specifies that the name of the transaction request is returned. If the returned value is CNMQUISE, it is a quiesce transaction generated by the CNMETQU service routine already invoked by the database server.

TRTYPE
Is a value indicating the type of transaction request that is returned. The following are the possible values:

QERR Indicates that this is a request transaction, but only error responses are to be returned to the requesting task.

QALL Indicates that this is a request transaction, but all operational and error response data is to be returned to the requesting task.

QNON Indicates that this is a request transaction, but that no responses (error condition or operational) are to be returned to the requesting task.

CORR Specifies that the correlator of the transaction request is returned. If all blanks are returned, the transaction request does not contain a correlator value.

VOCAB
Specifies that the context of parameter names is returned. If all blanks are returned, the default context is used.

ORIGNET
Specifies that the origin network ID of the transaction request is returned.

ORIGDOM
Specifies that the origin domain ID of the transaction request is returned.

ORIGTASK
Specifies that the origin task ID of the transaction request is returned. Normally this is the task ID of the NetView program requesting OST or PPT. However, if the received transaction request is a quiesce transaction, the *tiinq* name of this database server (as defined in CNMETIN) is returned.

DESTNET
Specifies that the destination network ID of the transaction request is returned.

DESTDOM
Specifies that the destination NetView domain ID of the transaction request is returned.

DESTTASK
Specifies that the destination task ID of the transaction request is returned. The value returned should be the NetView autotask ID that serves this database server.

gpnamep
Is a 31-character varying-length field that has two different uses:

- When you specify P with *gpmethod*, this field contains the name of the parameter whose value is being retrieved.
- When you specify N with *gpmethod*, this field is used to receive the name of the next parameter. The search for the next parameter begins at the location pointed to by *gpcursor*.

This field is required if you specify P or N in *gpmethod*. It is not required if you specify H.

gpindex
Contains a signed, 4-byte integer that has two different uses:

- When you specify P with *gpmethod*, this field contains the index value of the array parameter whose value is being retrieved. You must specify an index value of –32,767 if the parameter is not an array parameter.
- When you specify N with *gpmethod*, this field is used to receive the index value of the next parameter. The search for the next parameter begins at the location pointed to by *gpcursor*. A value of –32,767 is returned if this parameter is not an array type parameter.

This field is required if you specify P or N in *gpmethod*. It is not required if you specify H.

gpdata
Is a varying-length character field containing the value of the parameter being returned.

If the parameter type value (*gptype*) returned is GPTYPE=X'00EE', indicating that this is an EBCDIC character string, this field contains the character string in either the PL/I character varying format or the C Dsivarch format. If the

parameter type value is GPTYPE=X'00BB', indicating that this is a binary integer, this field contains the length of the fixed binary integer variable that is used to carry the binary integer.

The first two bytes indicate the length of the fixed binary integer variable that was used to carry the integer in the transaction request. For example:

- In PL/I, the length is 4 for a FIXED BIN(31,0) variable, or 2 for a FIXED BIN(15,0) variable.
- In C, the length is 4 for a long integer variable, or 2 for a short integer variable.

The remaining 4 or 2 bytes contain the binary integer.

gpdatlen
Is a 4-byte integer that specifies the maximum length of *gpdata*.

If the value specified by *gpdatlen* is less than the length of the parameter value to be returned, the parameter value is truncated and returned in *gpdata*, and a return code of RET_DATA_TRUNC is generated. The full length of the parameter value that was truncated is stored in HLBLENG (Hlbleng).

If the value specified by *gpdatlen* is equal to or greater than the length of the parameter value to be returned, and return code RET_GOOD is issued, the length of the returned parameter value is stored in HLBLENG (Hlbleng).

gptype
Is a 2-byte, fixed-length field that indicates the type of the parameter value returned in *gpdata*. One of the following values is set when this procedure returns:

X'00EE'
The returned parameter value is an EBCDIC character string.

X'00BB'
The returned parameter value is an integer.

This field is required if you specify P or N in *gpmethod*. It is not required if you specify H.

gpcursor
Is a 4-byte integer field used to keep track of the position in the scan. When you specify N in *gpmethod*, it must be initialized to a binary zero before the first call to CNMEGTP. Use of this field with a nonzero value can cause unpredictable results. This field is updated each time the service routine is called.

This field is required if you specify N in *gpmethod*. It is not required if you specify H or P.

Return Codes

The return codes for this service routine are:

Return Code Name	Value	Description
RET_GOOD	0	Successful.
RET_END_FILE	04	No more parameter names or values to retrieve (GPMETHOD=N).
RET_BAD_LENGTH	16	The *gpdatlen* specified contains an invalid value.

Return Code Name	Value	Description
RET_BAD_NAME	32	The requested *gpnameh* was invalid (GPMETHOD=H), or the requested *gpnamep* did not exist in the transaction (GPMETHOD=P).
RET_NO_DATA	40	The specified *gpnemeh* is valid, but either no value was found (GPMETHOD=H), or the specified *gpnamep* contains a length value of zero (GPMETHOD=P).
RET_PROCESSING_ERROR	48	Processing error during decoding.
RET_BAD_CURSOR	76	Specified cursor value is not valid (GPMETHOD=N).
RET_BAD_METHOD	80	The specified *gpmethod* is not valid.
RET_DATA_TRUNC	88	The parameter name or value is too long for the *gpdata* field provided for the returned data. Truncation has occurred. The actual length of the returned data is contained in HLBLENG.

Parameter Usage Reference

Table 6 summarizes the content of fields required for using the *gpmethod* parameter of the CNMEGTP service routine.

Table 6. Content Fields for gpmethod Parameter of CNMEGTP Service Routine

Parameter Name	Used when *gpmethod*=	Input/Output	Description
gpmethod	P,N,H	Input	Decoding method to use.
gpnameh	H	Input	Name of the header parameter.
gpnamep	P	Input	Name of the requested transaction parameter.
	N	Output	Name of the next transaction parameter.
gpindex	P	Input	Index of specified parameter.
	N	Output	Index of the next parameter.
gpdata	P,N,H	Output	Area to receive requested information.
gpdatlen	P,N,H	Input	Length of the provided data area.
gptype	P,N	Output	Type of parameter value.
gpcursor	N	Input/ Output	Position to scan for the next transaction parameter.

CNMENTR (CNMETNEXT) Service Routine—Get Next Transaction Request

The CNMENTR service routine receives the next transaction from the *tiinq* queue. After the transaction request is received, the parameters can be examined with calls to CNMEGTP.

The syntax for the CNMENTR command is:

PL/I CALL FORMAT:

CALL CNMENTR(*bhlbptr*)

PL/I MACRO FORMAT:

CNMETNEXT

C INVOCATION:

void Cnmentr()

Where:

bhlbptr
Is a fullword field containing the anchor pointer used by the server support API.

Return Codes

The return codes for this service routine are:

Return Code Name	Value	Description
RET_GOOD	0	Successful.
RET_NOT_FOUND	20	No transaction request available in queue *tiing*.
RET_STORAGE_FAILURE	52	Could not obtain working storage.
RET_PPI_FAILURE	1000 +*n*	When the routine invoked the program-to-program interface (PPI) module, it received PPI failure return code *n*.

CNMERTR (CNMETREADY) Service Routine—Ready For Next Transaction Request

The CNMERTR service routine sends a READY token to the NetView autotask that serves this database server. This token indicates that the server is ready to process another transaction request.

The syntax for the CNMERTR command is:

PL/I CALL FORMAT:

CALL CNMERTR(*bhlbptr*)

PL/I MACRO FORMAT:

CNMETREADY

C INVOCATION:

void Cnmertr()

Where:

bhlbptr
Is a fullword field containing the anchor pointer used by the server support API.

Return Codes

The return codes for this service routine are:

Return Code Name	Value	Description
RET_GOOD	0	Successful.
RET_BAD_STATE	24	Quiesce transaction already issued.
RET_PROCESSING_ERROR	48	Processing error occurred while encoding the READY token.
RET_STORAGE_FAILURE	52	Could not obtain working storage.
RET_PPI_FAILURE	1000 +*n*	When the routine invoked the PPI module, it received PPI failure return code *n*.

CNMESTR (CNMSENDSTR)—Send Transaction Reply to NetView Requester

The CNMESTR service routine sends a transaction reply to the requester in response to a transaction request.

The syntax for the CNMESTR command is:

PL/I CALL FORMAT:

CALL CNMESTR(*bhlbptr,transid,trtype,destnet,destdom,desttask, orignet,origdom,origtask,vocab,corr,respcode,parms*)

PL/I MACRO FORMAT:

CNMSENDSTR TRANSID(*transid*) TRTYPE(*trtype*) DESTNET(*destnet*) DESTDOM(*destdom*) DESTTASK(*desttask*) ORIGNET(*orignet*) ORIGDOM(*origdom*) ORIGTASK(*origtask*) VOCAB(*vocab*) CORR(*corr*) RESPCODE(*respcode*) PARMS(*parms*)

C INVOCATION:

void Cnmestr(char **transid*,char **trtype*,char **destnet*, char **destdom*,char **desttask*,char **orignet*, char **origdom*,char **origtask*,char **vocab*, char **corr*,char **respcode*,Dsicprm ***parms*)

Where:

bhlbptr
Is a fullword field containing the anchor pointer used by the server support API.

transid
Is the 8-character name of the transaction being executed. If the value of this field is less than 8 characters, it must be left-justified and padded with blanks. No embedded blanks are allowed in this value. The value of this field is obtained from the TRANSID field of the CNMEGTP service routine.

trtype
Is a 4-character parameter that indicates the transaction processing that was performed. The only valid value for this field is RPLY, which indicates that this is a reply in response to a request.

destnet
Is a fixed-length 8-character field that identifies the network that originated the transaction request. If the value of this field is less than 8 characters, it must be left-justified and padded with blanks. No embedded blanks are allowed in this value. The value for this field is obtained from the ORIGNET field of service routine CNMEGTP.

destdom
Is a fixed length 8-character field that identifies the domain that originated the transaction request. If the value of this field is less than 8 characters, it must be left-justified and padded with blanks. No embedded blanks are allowed in this value. The value for this field is obtained from the ORIGDOM field of service routine CNMEGTP.

desttask
Is a fixed-length 8-character field that identifies the task ID of the NetView OST or PPT that originated this transaction request. If the value of this field is less than 8 characters, it must be left-justified and padded with blanks. No imbedded blanks are allowed in this value. The value for this field is obtained from the ORIGTASK field of service routine CNMEGTP.

orignet
Is a fixed-length 8-character field that identifies the network ID of the network that is processing this transaction. If the value of this field is less than 8 characters, it must be left-justified and padded with blanks. No embedded blanks are allowed in this value. The value for this field is obtained from the DESTNET field of service routine CNMEGTP.

origdom
Is a fixed-length 8-character field that identifies the originating domain ID of the domain that is processing this transaction. If the value of this field is less than 8 characters, it must be left-justified and padded with blanks. No embedded blanks are allowed in this value. The value for this field is obtained from the DESTDOM field of service routine CNMEGTP.

origtask
Is a fixed-length 8-character field that identifies the task ID of the autotask that serves this database server. If the value of this field is less than 8 characters, it

must be left-justified, and padded with blanks. No embedded blanks are allowed in this value. The value for this field is obtained from the DESTTASK field of service routine CNMEGTP.

vocab
Is a fixed-length 8-character field that identifies the context that is used to interpret the parameter names in the transaction request. The value of this field is obtained from the VOCAB field of service routine CNMEGTP.

corr
Is a fixed length 8-character field that associates a response with any data that was kept by the requester at the time the transaction was issued. The value of this field is obtained from the CORR field of service routine CNMEGTP.

respcode
Is an 8-character fixed-length field that is used to summarize the results of a transaction request that has been processed. If no response code is required, the value of this field should be set to blanks.

parms
Contains a pointer to a linked list of parameter blocks. If no parameters are to be sent with the reply, this field should be set to either a C null (X'00000000') or a PL/I null (X'FF000000'). The following is the structure of a parameter block:

prmlink
Contains a pointer to the next parameter block in the chain. If this is the last parameter in the chain, this field should be set to either a C null (X'00000000') or a PL/I null (X'FF000000').

prmnaml
Contains a halfword binary integer that defines the length of the *prmname* field. This value must be in the range 1—31.

prmname
Is a fixed-length 31-character field that contains the name of the parameter.

prmindex
Contains a signed halfword binary integer that identifies the index value within a parameter that happens to be an array. A value of minus 32,767 is interpreted to mean that this parameter is not an array. Parameter arrays are intended to handle multiple values for a given parameter (for example, lines of text in a problem description).

prmtype
Contains a halfword binary integer that identifies the type of parameter. There are two possible values for this field:

X'00EE'
Indicates that the parameter value is an EBCDIC character string.

X'00BB'
Indicates that the parameter value is a binary integer.

prmleng
Contains a halfword binary integer that identifies the length of the parameter value. The value of this field must be less than 31,000.

If the parameter type value (*prmtype*) returned is PRMTYPE=X'00EE', indicating that this is an EBCDIC character string, this field contains the length of the character string in the number of bytes. If the parameter

type value is PRMTYPE=X'00BB', indicating that this is a binary integer, this field contains the length of the fixed binary integer variable that is used to carry the binary integer. For example:

- In PL/I, the length is 4 for a FIXED BIN(31,0) variable, or 2 for a FIXED BIN(15,0) variable.
- In C, the length is 4 for a long integer variable, or 2 for a short integer variable.

Note: The actual maximum length of any parameter value depends on the length of all fields in the transaction. The maximum length of any transaction is 31K. For important information on calculating the size of a transaction, see "Understanding Transaction Size Limitations" on page 55.

If a transaction reply is longer than the maximum length, it is truncated and sent. A CNM_TRUNCATION parameter with the EBCDIC value YES is placed at the end of the reply to indicate the truncation to the NetView Requester API receiver.

prmptr Contains a pointer to the value of the parameter. If you do not specify a value, this field should be set to either a C null (X'00000000') or a PL/I null (X'FF000000').

Note: The NetView program provides the mapping of the parameter block in PL/I and C under the include file names DSIPPRM (for PL/I), and DSICPRM (for C). These files are included by DSIBPLI and DSIBC. Therefore, all you need to do is include DSIBPLI or DSIBC into your program.

Return Codes

The return codes for this service routine are:

Return Code Name	Value	Description
RET_GOOD	0	Successful.
RET_BAD_TRAN_HDR	36	One of the transaction header parameters (for example, *transid, trtype, destnet, destdom, desttask, orignet, origdom*, or *origtask*) contains an invalid value.
RET_PROCESSING_ERROR	48	Processing error occurred while encoding the transaction reply.
RET_STORAGE_FAILURE	52	Could not obtain working storage.
RET_TRAN_REPLY_TRUNC	60	Transaction reply was truncated and sent.
RET_PPI_FAILURE	1000 +*n*	When the routine invoked PPI module it received PPI failure return code *n*.
RET_BAD_TRAN_PARM	10000 +*n*	The *n*th parameter block in the link list pointed to by the PARMS contains either an invalid *parmnaml* or *prmleng* value, or both.

CNMETIN (CNMETINIT) Service Routine—Initialize the Server Support API

The CNMETIN service routine is called once to initialize the database server.

The syntax for the CNMETIN command is:

PL/I CALL FORMAT:

CALL CNMETIN(*bhlbptr,tiinq,tiinqlim,tioutq*)

PL/I MACRO FORMAT:

CNMETINIT INQUEUE(*tiinq*) INQUEUELIM(*tiinqlim*) OUTQUEUE(*tioutq*)

C INVOCATION:

void Cnmetin(*char *tiinq,int tiinqlim,char *tioutq*)

Where:

bhlbptr
Is a fullword field that contains the anchor pointer on return from CNMETIN. The anchor pointer is required as an input parameter by all other service routines.

tiinq
Is an 8-character fixed-length field containing the name of the program-to-program interface (PPI) queue that is used to receive transaction requests from the NetView program. This field must contain 1-8 characters. These characters can be uppercase alphabetic (A—Z), numeric (0—9), or $, %, &, @, and #. If this field does not contain 8 characters, it must be left-justified and padded with blanks.

tiinqlim
Is a 4-byte integer field containing the limit of the PPI *tiinq* queue. This limit cannot be a negative value. The recommended value for this field is the number of ready tokens that are issued by this database server, multiplied by four.

tioutq
Is an 8-character fixed-length field containing the name of the PPI queue to which transaction replies are sent.

Note: This is the same name that you assigned to the *oqueue* parameter of the RTRINIT command (see “RTRINIT Command” on page 87).

This field must contain 1-8 characters. These characters can be uppercase alphabetic (A—Z), numeric (0—9), or $, %, &, @, and #. If this field does not contain 8 characters, it must be left-justified and padded with blanks.

Return Codes

The return codes for this service routine are:

Return Code Name	Value	Description
RET_GOOD	0	Successful.
RET_BAD_TIINQLIM	28	Specified *tiinqlim* contains a negative value.

Return Code Name	Value	Description
RET_STORAGE_FAILURE	52	Could not obtain storage for the required buffers that would be used by the server support API. Initialization failed.
RET_LOAD_FAILURE	84	The initialization routine was unable to load PPI module CNMNETV. Initialization failed.
RET_PPI_FAILURE	1000 +*n*	When the routine invoked PPI module, it received PPI failure return code *n*. Initialization failed.

CNMETQU (CNMETQUIESCE) Service Routine—Quiesce This Database Server

The CNMETQU service routine is called to initiate an orderly shutdown of the database server issuing the call. Other database servers remain active. After CNMETQU is called, no further calls to CNMERTR (ready for next transaction request) are accepted.

The database server continues processing until the quiesce transaction is returned from the NetView Bridge dispatcher. At that time, the database server calls CNMETRM (terminate the server support API).

The quiesce transaction that the NetView program receives from this service routine can be identified by checking the TRANSID field (it must contain a value of CNMQUISE). In addition, the ORIGTASK field contains the database server's *tiinq* name (as defined in service routine CNMETIN).

The syntax for the CNMETQU command is:

PL/I CALL FORMAT:

CALL CNMETQU(*bhlbptr*)

PL/I MACRO FORMAT:

CNMETQUIESCE

C INVOCATION:

void Cnmetqu()

Where:

bhlbptr
Is a fullword field containing the anchor pointer used by the server support API.

Return Codes

The return codes for this service routine are:

Return Code Name	Value	Description
RET_GOOD	0	Successful.

Return Code Name	Value	Description
RET_BAD_STATE	24	Quiesce transaction already issued.
RET_PROCESSING_ERROR	48	Processing error occurred while encoding the quiesce transaction.
RET_STORAGE_FAILURE	52	Could not obtain working storage.
RET_PPI_FAILURE	1000 +*n*	When the routine invoked the PPI module, it received PPI failure return code *n*.

CNMETRM (CNMETTERM) Service Routine—Terminate the Server Support API

The CNMETRM service routine allows you to disconnect the database server from the NetView program and clean up the API control blocks. No further API calls can be issued after this service routine is called, unless you reinitialize the server by calling CNMETIN.

The syntax for the CNMETRM command is:

PL/I CALL FORMAT:

CALL CNMETRM(*bhlbptr*)

PL/I MACRO:

CNMETTERM

C INVOCATION:

void Cnmetrm()

Where:

bhlbptr
Is a fullword field containing the anchor pointer used by the server support API.

CNMEWAT (CNMETWAIT) Service Routine—Wait for a Transaction Request

The CNMEWAT service routine is called to wait for work from either the input queue (from the NetView program) or from some other source (the posting of an event control block).

The syntax for the CNMEWAT command is:

PL/I CALL FORMAT:

CALL CNMEWAT(*bhlbptr,waecbp*)

PL/I MACRO FORMAT:

CNMETWAIT ECBLIST(*waecbp*)

C INVOCATION:

void Cnmewat(*void *waecbp*)

Where:

bhlbptr
Is a fullword field containing the anchor pointer used by the server support API.

waecbp
Contains the pointer to a list of event control blocks (ECBs) that should be included in the list for which CNMEWAT is waiting. The ECB for work waiting from the *tiinq* queue (from the NetView program) is provided by the service routine, and is not included in this ECB list.

An example of an ECB that should be included is the termination ECB. This ECB allows the server to terminate the wait state should some other event (such as an MVS STOP command) occur before a transaction is received.

If no ECBs are to be included in the wait list, this field should be set to a C or PL/I null value. If specified, the ECB list contains a contiguous list of fullword addresses of the appropriate ECBs. The last entry in the list of ECB addresses has its high-order bit (0) set to 1 to indicate the end of the list.

Return Codes

The return codes for this service routine are:

Return Code Name	Value	Description
RET_WAIT_POSTED_PPI	8	PPI *tiinq* ECB is posted. One or more transaction requests are available. If you have specified your own ECB list in WAECBP, check to see if the ECBs in the list have also been posted.
RET_WAIT_POSTED	12	One or more user-provided ECBs have been posted. The PPI *tiinq* ECB is not posted and no transaction request is available.
RET_STORAGE_FAILURE	52	Could not obtain working storage.

Server Support API Debugging Support

The server support API service routines maintain an 8-entry continuously wrapping trace area. The name of this in-storage trace area is HLBFFOCA. It is located in control block DSIBHLB. You can print the contents of the trace area during execution by including appropriate print instructions in your PL/I or C program.

Trace entries are recorded at entry to and exit from all service routines. In the event of an abnormal end (abend), this area indicates which server support API module was running and either the address of the calling routine or the return code from the service routine.

Each entry in the trace area is 6 bytes in length, making the total size of the trace area 48 bytes. Each trace point is identified by a unique 16-bit ID number. The first 12 bits represent the module ID and the next 4 bits represent the location ID within the module.

The following are the hexadecimal values and corresponding modules for the module IDs:

Hex Value	Module
X'051'	CNMETIN
X'052'	CNMETRM
X'053'	CNMEWAT
X'054'	CNMEGTP
X'055'	CNMENTR
X'056'	CNMETQU
X'057'	CNMERTR
X'058'	CNMESTR

The following are the hexadecimal values and corresponding debug points for the location IDs:

Hex Value	Debug Point
X'0'	Module entry
X'F'	Module exit

The next 4 bytes of the trace entry contain debugging information captured at the debugging point. If the error occurred upon entry, these 4 bytes contain the address of the caller (from register 14 of the caller). If the error occurred during exit, these 4 bytes contain the return code from the service routine.

Appendix D. NetView Bridge Messages and Abend Codes

This appendix lists the messages and abend codes that are used with the NetView Bridge function.

Messages for the NetView Bridge

The NetView Bridge uses several messages during normal processing. Some messages are passed across the bridge with data attached to them and are used primarily for constructing transaction requests and replies. Use the NetView program's online help for additional information on the NetView Bridge messages.

The NetView Bridge messages are:

- DWO150I
- DWO151I
- DWO152I
- DWO156I
- DWO157I
- DWO530I
- DWO531I
- DWO533I
- DWO534I
- DWO535I
- DWO536I
- DWO537I
- DWO539I
- DWO546I
- DWO548I
- DWO550I
- DWO568I
- DWO569I

Abend Codes for the NetView Bridge

An abend code is generated for an unrecoverable error. This completion code is issued when an internal NetView logic error occurs or when the NetView program detects a serious problem. The code is presented on an abend dump listing as user code U*xx*, where *xx* is the decimal code as described in this section.

Table 7. Abend Codes for NetView Bridge

Decimal Abend Code	Hex Abend Code	Explanation of the Abend
85	X'55'	The initialization routine could not obtain storage for the primary anchor control block whose address would be returned in the *bhlbptr* parameter of CNMETIN (CNMETINIT). Initialization of the server support API failed.
86	X'56'	The initialization routine could not load either of the required NetView load modules DSIEHLAR, DSIEHL24, or both. Initialization of the server support API failed.
91	X'05B'	Should not occur; severe error returns from the NetView Bridge dispatcher or remote dispatcher.

A task that has abended the maximum number of times (MAXABEND) no longer processes transactions. Refer to “Documenting and Reporting Problems” in *Tivoli NetView for OS/390 Diagnosis Guide* for recovery procedures.

Index

Special Characters

A

B

C

D

E

F

H

I

J

L

M

N

O

P

Q

R

S

T

U

W

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC31-8238-03