CA Mainframe Network Management

Network Control Language Programming Guide

Release 12.1

This Documentation, which includes embedded help systems and electronically distributed materials, (hereinafter referred to as the "Documentation") is for your informational purposes only and is subject to change or withdrawal by CA at any time.

This Documentation may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA. This Documentation is confidential and proprietary information of CA and may not be disclosed by you or used for any purpose other than as may be permitted in (i) a separate agreement between you and CA governing your use of the CA software to which the Documentation relates; or (ii) a separate confidentiality agreement between you and CA.

Notwithstanding the foregoing, if you are a licensed user of the software product(s) addressed in the Documentation, you may print or otherwise make available a reasonable number of copies of the Documentation for internal use by you and your employees in connection with that software, provided that all CA copyright notices and legends are affixed to each reproduced copy.

The right to print or otherwise make available copies of the Documentation is limited to the period during which the applicable license for such software remains in full force and effect. Should the license terminate for any reason, it is your responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in the Documentation is governed by the applicable license agreement and such license agreement is not modified in any way by the terms of this notice.

The manufacturer of this Documentation is CA.

Provided with "Restricted Rights." Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

Copyright © 2010 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Contact CA Technologies

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

Contents

Chapter 1: Introduction

This section contains the following topics:

About this Guide

This guide provides information for anyone who writes or maintains NCL procedures. It describes the principles behind NCL and includes reference material and examples. It describes how NCL is structured and used. It describes important components of the language and how these components are used within NCL statements to drive your products.

The components (verbs, built-in functions, and system variables) are listed. The manipulation of NetMaster® Databases (NDBs) is described with step-by-step instructions for creating an NDB and examples of code to demonstrate how best to work with NDBs.

What You Need to Know Before Using NCL

This guide assumes that your product is already installed. It assumes that you are already familiar with basic IBM computer concepts and terminology. Specific NCL concepts and terminology are described where appropriate in the text, usually when first mentioned.

Related Documentation

This guide and the *Network Control Language Reference Guide* are the principal reference documents for users of NCL.

The *Network Control Language Reference Guide* contains descriptions of all NCL and &NDB verbs, built-in functions, and system variables, their syntax and usage.

Note: For more information about administering and using the core functions of your products, see the following resources:

- *Administration Guide*
- *Reference Guide*
- *Security Guide*
- *User Guide*
- Online Help

Chapter 2: About Network Control Language

This section contains the following topics:

What Is Network Control Language (NCL)?

Network Control Language (NCL) is a high-level interpretive language integrated into your products. It provides a fast, comprehensive and advanced development tool to implement your site-specific requirements. You can use NCL to rapidly tailor your product to suit the needs of your environment. NCL is based on free-form statement syntax that can process both system and user-supplied data. Data is maintained in variables that can be manipulated and changed as required.

Collections of NCL statements, which can include system commands, are termed procedures, and are stored in partitioned data sets (OS/VS) or CMS files (VM) called procedure libraries. These procedure libraries can be edited and updated while your system is operational. Each NCL procedure is a separate member within a procedure library.

There is one principal procedure library (or concatenation of libraries) used by your system. In addition to this system library, individual users under OS/VS can be allocated an individual procedure library for their own use, as part of the definition of their user ID.

NCL procedures can take many forms. They can be a simple collection of comment statements that provide an effective means of online documentation. They can be a collection of product commands in exactly the same format as entered from a terminal. They can be extended to include logical decision-making capabilities, the display of full-screen panels, and the use of file processing capabilities.

An NCL procedure can call or nest another procedure to improve modularity.

In addition, certain NCL procedures are reserved for performing special functions such as interfacing to unsolicited messages from VTAM (PPOPROC), intercepting and reacting to other messages sent to the user terminal (MSGPROC), and processing messages destined for the activity logs (LOGPROC).

Create NCL Procedures

Procedures can be created while the system is operational. New procedures can be added to the procedure library, or existing procedures modified. Any changes made to a procedure become effective the next time the procedure is loaded.

Procedures can be created using either system utilities or an online editor such as TSO/ISPF or CMS Edit.

To create a new procedure you must first select a (unique) name by which the procedure is to be identified and by which it will later be executed. The name you select should be meaningful and where possible identify the type of function that the procedure performs. CA recommends that you establish naming conventions for this library to ensure consistent use of names across operational areas.

Procedures are created as a series of individual records. Although the format of statements within each record is flexible, certain syntax restrictions do apply.

Commands, comments, and statements can be entered in either upper or lower case. Most commands are converted to upper case before execution. Comments are left unchanged.

Check NCL Syntax

When you have written an NCL procedure, you can use the NCLCHECK command to verify the syntax and structure without executing the procedure. NCLCHECK loads and checks the procedure just as if it were to be executed and notifies you of major syntax or structural errors in your code. Full syntax checking does not take place until execution.

Test NCL Procedures

To ensure that system overhead is kept to a minimum, the system attempts to optimize NCL execution in many ways. One technique is that of sharing concurrent requests for a procedure of the same name, in such a way that only one copy of the procedure is loaded into storage. This technique also lets you specify a retention queue on which a completed procedure is retained, on the assumption that it can be reused within a short period. This eliminates the need to perform disk I/O to bring the procedure into storage again.

For extremely high usage procedures, you can use a technique known as preloading to ensure that the procedure remains in storage at all times, regardless of its pattern of use. These options are selected by the LOAD command.

Test in Production and Testing Environments

While such facilities are ideal in a production environment they can interfere with testing. For example, you may have completed and tested a procedure and made certain corrections and want to run a second test. Although you have saved the corrected procedure in its library, you may notice the original errors are still occurring. This is most likely to be caused by NCL having retained the original procedure in storage and reused it when the second test was run. To overcome this apparent interference with your testing, use the NCLTEST command to signal to the system that you are in fact operating in a test mode and do not want your procedures retained or shared and that the latest copy is always to be loaded from the library. For more information about the NCLTEST command, see the Online Help.

Debug an NCL Procedure

The NCL Debug facility is a powerful tool to assist in the debugging of NCL procedures.

NCL Debug provides the ability to observe the execution of an NCL procedure from an external source, that is, another environment, another user region, window, or NCL environment. It eliminates the need for code changes to debug a procedure. It also provides comprehensive control over the NCL procedure as it is being executed, supporting statement stepping, alteration of variable contents and attributes, and so on. It lets you specify criteria for debugging NCL before it begins execution.

The NCL Debug facility is made up of a set of commands that lets you start and stop an NCL debug session, control the execution of NCL processes, and display and modify the contents of NCL variables. Procedure and subroutine nesting levels can be listed and the source code that is being executed can be displayed. NCL trace output can be received at another region, thus allowing you to view the trace output concurrently as the debugged process executes.

Invoke and Cancel NCL Procedures

NCL procedures are invoked and canceled by using commands.

Invoke NCL Procedures

Procedures are invoked explicitly by using the EXEC and START commands, or implicitly by certain system functions such as EASINET and timer commands.

Optionally, variable parameters can be passed to the procedure when it is invoked. These variables are specified on the EXEC or START command following the name of the procedure being invoked. There is no limit to the number of variables that can be passed. Each word following the procedure name is passed in the next available variable, numbering from &1 for the first word, &2 for the second, and so on.

In VM systems the CMS minidisk that holds your product procedures must be reaccessed after the procedures have been altered, so that your product locates the new versions of the changed procedures.

Cancel NCL Procedures

NCL procedures can be canceled by using the FLUSH or END commands. Generally users can cancel only their own procedures, but the system can be configured so that authorized users can cancel any procedure. Full-screen procedures (that is, procedures that display full-screen panels) can normally be canceled by using the standard END function keys (F3/4 or F15/16). If function key interception is being performed by the procedure, the responsibility of responding correctly to the use of function keys rests with the author of the procedure.

Exit OCS During Execution

When an Operator Console Services (OCS) operator exits OCS, any NCL processes still executing in their OCS window are flushed without further execution and any display lines awaiting output are discarded. Any processes that are queued for execution by that user are also flushed. Messages are written to the log identifying any procedures that have been flushed.

Control Runaway Loops

Protection against uncontrolled looping within a procedure is provided through a loop control counter, maintained automatically by NCL. Runaway loop control can be activated by using &CONTROL LOOPCHK verb. The default is &CONTROL NOLOOPCHK, which means that runaway loop control is not in effect unless requested.

When a procedure commences execution, a count of 1000 is assigned to this counter. This is then decremented by one for each executed &DOEND statement associated with an &DOWHILE or &DOUNTIL, and for each executed &GOTO statement. If the count reaches zero, the procedure is regarded as being in an uncontrolled loop and is automatically terminated.

Certain events that involve operator interaction, indicating that the procedure is not looping, cause this counter to be automatically reset. An example of this occurs when displaying a panel using the &PANEL statement and then having to wait for operator input before processing can continue. In such a case the value for the loop control counter is reset to 1000. The &LOOPCTL verb is provided to enable the user to set a new loop control limit from within the procedure for those procedures that require more than the default value.

Note: &LOOPCTL is decremented for PPOPROC, MSGPROC, LOGPROC, AOMPROC, or CNMPROC. However, it is reset to its full value each time a message is read with &PPOREAD, &MSGREAD, &LOGREAD, &AOMREAD, &CNMREAD, or &INTREAD. The loop control therefore applies only for processing associated with one message.

List Procedure Names

Note: This applies to OS/VS only.

The SHOW EXEC command can be used to obtain an online list of the names of procedures in the procedure library. Optionally, a specified range can be listed:

```
SHOW EXEC,CA,CD
```

or the procedures in a particular procedure library can be listed:

```
SHOW EXEC,ID=PROCLIB2
```

For more information about the SHOW EXEC command, see the Online Help.

List the Contents of NCL Procedures

The LIST command displays the contents of the nominated procedure. No statements or commands within the procedure are executed. Nested procedures are not listed, and separate LIST commands are required if these need to be displayed.

Chapter 3: NCL Concepts

This section contains the following topics:

Where Does NCL Execute?

NCL executes only in association with a real or internally-simulated terminal. The most flexible method of invoking NCL processing, and the one that is easiest to understand, is to use the EXEC or START commands from an OCS window. These commands let you specify the name of the NCL program that you want to run.

Other methods of executing NCL are implicit (see page 36) rather than explicit.

What Is an NCL Procedure?

The basic unit of NCL code is called an NCL procedure. A procedure contains one or more NCL statements that are executed when the procedure is invoked.

Each NCL procedure has a 1- to 8-character name and resides in your product's procedure library. In OS/VS systems this is a Partitioned Data Set (PDS). In VM systems the library is held on a CMS minidisk.

Procedures are created or modified by an appropriate editor, for example, ISPF.

What Is an NCL Process?

The actual execution of an NCL procedure is termed an NCL process. The EXEC or START command specifies the name of an NCL procedure, which is either retrieved from the NCL procedure library or is already resident in storage.

The following example shows the logical difference between a procedure and a process:

```
START      PROC1   NCPA
START      PROC1   NCPB
START      PROC1   NCPC
```

These commands, entered from an OCS window, invoke three separate NCL processes, each of which is executing procedure PROC1. There can be only one copy of PROC1 NCL code in storage, but three separate executions of it are taking place concurrently. In this example each process is given a different NCP name as a parameter.

It is nevertheless common for the term procedure to be used instead of process.

Nesting

The first procedure that is executed in a process can issue EXEC commands internally to call other procedures. This is called nesting. Regardless of how many other procedures are called, or the number of nesting levels involved, all execute as part of the same process. If any procedure in a process issues a START command, a new process is invoked, which is independent of the originating process.

Execution of the procedures in a process continues according to the logic of the procedures. When the first level procedure completes its processing, it ends and the process terminates.

NCL Process Identifier

Each NCL process has a unique number associated with it when it starts execution. This is the NCL Process Identifier (NCLID), and it is a 6-digit number in the range 1 to 999999. Processes executing at the same time carry different identifiers.

The process identifier lets individual processes be identified, even when processes are executing the same procedure.

The process identifier is used principally to specify which process is to be the target of a command such as GO, FLUSH, or INTQ, all of which can talk to NCL processes.

NCL Processing Region

All NCL processing that occurs within a system is performed on behalf of product users. These users can be people, who log on to the system with a user ID/password combination. Also, there are internal environments that act like real users, except that they do not have any real terminal associated with them. These internal environments all have virtual user IDs and in fact can be profiled in the same way as a real user ID by having a UAMS definition created.

The most common virtual user IDs are those associated with the background environments known as the background logger and background monitor. If you issue a SHOW USERS command from an OCS window, you will see these virtual user IDs listed along with any real users logged onto the system at the time. Other optional product components generate their own background environments, so the list that you see on a SHOW USERS display varies according to the configuration of the system.

Regardless of whether a user is real or virtual, every user in a product region is capable of executing NCL processes, because every user has an NCL processing region associated with their user ID while they are logged on.

The NCL processing region provides all the internal services necessary to allow the user to have NCL processes executed on their behalf. While there is one NCL processing region for each user; within each user region there can be one or more NCL processing environments.

NCL Processing Environment

Each real user is associated with a 3270-type terminal with a display screen that supports either one or two logical windows at any time.

Virtual users have no real terminals associated with them, but operate logically as if they own one real line mode window.

Associated with each window that a user operates is at least one primary processing environment. A processing environment provides the internal services and facilities that are required to execute NCL processes for the user from its associated window.

Real users, using real terminals, can have one or two windows and therefore can have one or two active NCL processing environments directly associated with these windows. Internal users have only one window (logically operating in line mode) and therefore have only one NCL processing environment.

An NCL process always operates within the NCL processing environment associated with the window from which it was invoked.

Execute NCL Processes Serially

You can use the EXEC command to invoke an NCL process which will execute serially with respect to other NCL processes that are invoked by the EXEC command from the same window (that is, in the same processing environment).

This means that if you enter:

```
EXEC     PROC1
EXEC     PROC2
```

from the OCS window command line, a process is invoked to execute procedure PROC1 immediately; the process invoked to execute PROC2 waits until the first process ends before starting to execute. The use of the EXEC command therefore serializes execution of NCL processes in each NCL environment.

Serial execution of processes is useful for handling sequences of functions or operations.

Remember: Each NCL processing environment is independent; each one can be executing its own stream of serialized processes. There is no limit to the number of processes that can be queued for serial execution within an NCL processing environment.

Execute NCL Processes Concurrently

You can use the START command to invoke an NCL process that will execute concurrently with other NCL processes that are invoked with the EXEC or START command from the same window (that is, in the same processing environment).

This means that if you enter:

```
START     PROC1
START     PROC2
```

from the OCS window command line, a process is invoked to execute the procedure PROC1 immediately, and the second process is also invoked to execute PROC2 immediately. You can use the START command to execute many independent NCL processes at the same time in the same NCL environment.

Concurrent execution of processes lets you have slave processes running on your behalf doing different tasks. For example, a process could execute NCL procedures that monitor the status of particular resources at regular intervals and communicate with your OCS window only when something is found to be wrong.

The default maximum number of NCL processes that can be executing concurrently for any user is 128.

The concurrent stream of NCL processes executes in parallel with the serial stream.

If you enter:

```
EXEC     PROC1
EXEC     PROC2
START    PROC3
START    PROC4
```

from the OCS window command line, PROC1 executes at once on the serial stream and PROC2 is queued waiting for PROC1 to finish. However, PROC3 and PROC4 start at once on the concurrent stream and therefore execute at the same time as PROC1.

Dependent Processing Environment

Processing environments can operate in a hierarchy, so that there can be many processing environments associated with the same window. This is achieved by using the &INTCMD NCL verb.

Every NCL process executing within an NCL processing environment can establish a dependent processing environment (using the &INTCMD verb) in which other NCL processes can be invoked to execute on behalf of the higher level.

In turn, an NCL process executing in a dependent processing environment can establish its own dependent environment, forming a hierarchy of dependency and allowing any NCL process to invoke chains of other dependent processes on its behalf.

&INTCMD Verb

&INTCMD lets a procedure issue commands or execute other NCL processes and have the results returned to it rather than returning to the user's terminal window. This lets users write sophisticated procedures that check the results of their actions and correlate commands with their results.

When an &INTCMD statement is executed by a procedure, a new NCL processing environment is created that is subordinate to the process that owns it, that is, the process from which the &INTCMD statement is issued. This new processing environment is called a dependent processing environment, because it exists only until its originating process ends or issues an &INTCLEAR verb. The originating process can use &INTCMD to schedule commands or other NCL processes for execution within its dependent processing environment.

Any NCL process can issue &INTCMD and create its own dependent processing environment. As described before, if you issue the commands:

```
START    PROC1

START    PROC2
```

from your OCS window command line, processes PROC1 and PROC2 start executing in your window's NCL processing environment.

If each of those processes issues:

```
&INTCMD  START  PROC3
```

then PROC1 and PROC2 each have a dependent processing environment in which a process PROC3 is executing. Similarly, the PROC3 processes can issue their own &INTCMD statements to create and use their own dependent environments.

Any process that ends or that issues an &INTCLEAR statement, automatically causes the termination of all dependent environments, and therefore causes termination of whatever hierarchy of processes exists below it.

Explicit NCL Process Execution

The examples shown in the preceding sections have all assumed that the NCL processes involved have been executed as a result of a START or EXEC command. Using START or EXEC to invoke NCL processes is called explicit execution.

Implicit NCL Execution

Certain system functions execute NCL processes on behalf of users. The Primary Menu NCL procedure is an example of this. When a user logs on, the NCL procedure nominated on the SYSPARMS MENUPROC command is executed.

In this case, NCL execution has been invoked implicitly as a result of a user logging on, rather than as a result of a START or EXEC command.

Other examples of implicit NCL execution are:

- MAI menu procedure (MAI selected from the Primary Menu)
- EASINET procedure (invoked on behalf of each terminal when it connects)

System Level Procedures

Implicit NCL execution also occurs in relation to the various system level procedures that operate internally. The most common examples are:

LOGPROC

This process executes the NCL procedure nominated on the SYSPARMS LOGPROC= command. The process executes in the processing region of the LOGP user ID type, which is a virtual user. LOGPROC views, and can act on, all messages that flow to the log.

PPOPROC

This process executes the NCL procedure nominated on the SYSPARMS PPOPROC= command. The process executes in the processing region of the PPOP user ID type, which is a virtual user. PPOPROC views, and can act on, all unsolicited messages that are sent to your product by VTAM to report network events.

Other system level procedures can be started by optional components, for example, the Network Error Warning System (CNMPROC).

MSGPROC Procedure

A user operating in OCS mode can have a MSGPROC procedure associated with the window.

The MSGPROC procedure is invoked automatically when the user selects OCS mode from the Primary Menu, and is therefore implicitly executed. The MSGPROC procedure has access to all messages that are sent to the OCS window and can intercept, change or delete them as required. Message attributes such as color and highlight options can also be changed.

MSGPROC facilities are available to OCS windows running on real terminals, to background environments, system level procedures, and console environments. Processes that execute in dependent processing environments cannot have MSGPROCs associated with them.

Issue Commands from an NCL Process

Each NCL process executes within the NCL processing region of the user ID that invoked it and is entitled to execute any command the user can execute from an OCS window command line. The process therefore has the same command authority as the user for which it is executing.

If a process executes a command, the rules outlined in the following sections dictate where the results of the command are sent.

Inline Command Execution

If an OCS operator enters:

```
START    PROC1
```

from the OCS window command line, the process PROC1 starts running.

If the process then executes the command:

```
SHOW  USERS
```

the results of the command (the answer) flow to the OCS window. The process itself never has access to the messages generated as a result of the command. Consequently the procedure cannot make any decisions based on the results of the command.

The execution of a command by a process in this example is called inline command execution.

Dependent Command Execution

If PROC1 is started in the same way as described in the previous section but executes the NCL statement:

```
&INTCMD  SHOW  USERS
```

the SHOW USERS command is executed in the dependent processing environment for the process. The messages generated by the command are not returned to the user's OCS window but queued in a stack called the dependent response queue. The process PROC1 can then issue the NCL statement:

```
&INTREAD  ARGS
```

to read the results of the command from the dependent response queue one message at a time. The individual words of each message are tokenized and placed in variables of the form &1 &2... &n.

This technique lets a process issue a command and get the results back internally, so that the process can review the results and therefore make a decision based on the results of the command. This allows correlation of commands and results which in turn provides unlimited capability for complex logic to be built into processes to handle monitoring and automation of events.

The use of &INTCMD to execute commands privately in this way is called dependent execution of commands.

Review of Message Delivery Rules

The principal difference between inline and dependent command execution is the message delivery that applies to the two different techniques.

A process that issues an inline command never sees the results of the command; the results are always returned to the owner of the NCL processing environment in which the process is running.

A process that issues a dependent command sees the results of the command because they are queued to its dependent response queue and can be read using the &INTREAD statement.

NCL Processes and the Remote Operator Facility (ROF)

The concepts of NCL operation discussed so far have covered execution of processes within the NCL processing region of the user, in the system where the user is physically logged on.

ROF lets a user, who is logged on in one system, route commands to other systems for execution. The results of these commands are then returned to the originating user.

ROF also allows a command to be routed to one system for onward propagation to another system where the command is to be actually executed.

Since ROF provides services at a command level, NCL processes (which can issue commands) are also entitled to use ROF services to route commands to a remote system and retrieve the results.

Message Flow on a ROF Session

A user who issues a SIGNON command to another system or who issues a ROUTE command to send a command to another system for execution establishes a ROF session. The user is logically logged on to the remote system and has user attributes and privileges as defined to the remote system.

If you enter:

```
ROUTE SOL2 SHOW USERS
```

from the OCS window command line, the SHOW USERS command is sent to the remote system known as SOL2 and executed under your ROF logon. The results of the command are returned and displayed on your real OCS window.

If you enter:

```
START PROC1
```

to start the process PROC1, which in turn issues the inline command:

```
ROUTE SOL2 SHOW USERS
```

the results are also returned to the real OCS window. In other words, the delivery of the results of inline commands is the same across a ROF session as it is within the one system. If a process executes an inline command either in its own system or by routing the command to another system, the results return to the owner of the NCL processing environment in which the process is executing; they do not return to the process itself.

Alternatively, if the process in the example issues the statement:

```
&INTCMD ROUTE SOL2 SHOW USERS
```

the results return to the PROC1 dependent response queue, and can be read back by PROC1 using the &INTREAD statement.

In summary, therefore, the delivery of command results is always the same, regardless of whether a process issues a command for execution in its own system or a remote one.

The ability of NCL processes to issue commands across ROF sessions, and to correlate the results, allows the development of monitoring and control processes that can operate unseen and communicate with the operator only when a problem occurs.

Communication Between Processes

NCL processes execute in isolation from one another and although many processes can run concurrently in the same NCL environment or region, they usually have no knowledge of each other's existence.

However, it is often very useful to communicate directly with another NCL process either within the same NCL region or (depending on implementation options and user authority) across NCL regions.

The ability of processes to talk to each other provides the framework for developing co-operative processes that can coordinate their activities but remain independent. ROF services can also be used to provide communication between processes in different systems.

INTQUE Command

As described in the preceding sections, a process receives the results of commands that it issues using the &INTREAD verb. In concept there is a queue associated with the process, to which can be added messages that represent the results of commands. The &INTREAD statement allows the process to take messages off the queue one by one and process them as required.

The queue used for the messages generated by commands is called the dependent response queue.

The INTQUE command can also be used to place messages on a process's dependent response queue either by entering the command from an OCS window command entry line, or (more commonly) by issuing the INTQUE command from a different process.

Example: INTQUE command

The following example illustrates the concept. An OCS operator enters:

```
START  PROC1
```

from the OCS window, which starts the process PROC1.

PROC1 then executes the following statements:

```
&WRITE  DATA=&ZNCLID READY FOR WORK.
&INTREAD  ARGS
```

At this point the process suspends execution pending the arrival of message(s) on its dependent response queue.

The operator, or another NCL process, then enters:

```
INTQUE  ID=357  TYPE=RESP  DATA=BEGIN
```

where 357 is the NCL process identifier of the process PROC1 and TYPE=RESP indicates that the text of the command is to be placed on the PROC1 dependent response queue. The &INTREAD of the process then completes, with the variable &1 containing the word BEGIN. The process can then go ahead with whatever other processing is required, having used the &INTREAD and INTQUE combination to provide direct communication between the process and the operator.

Dependent Request Queue

While the INTQUE command can be used to place messages on the dependent response queue of a target process, it is sometimes confusing if messages arrive from an INTQUE command when the process is also expecting the results of a command it has issued. The INTQUE messages might arrive in the middle of the messages generated by the command.

To avoid this and to provide a method of isolating messages generated by commands issued by a process (which are predictable) from messages arriving from INTQUE commands issued externally (which are unpredictable), a second message stream is available. This is the dependent request queue.

Example: Dependent Request Queue

If the example in the previous section is changed so that PROC1 executes:

```
&WRITE DATA=&ZNCLID READY FOR WORK.
&INTREAD ARGS TYPE=REQ
```

and the operator enters:

```
INTQUE ID=357 TYPE=REQ DATA=BEGIN
```

then the &INTREAD issued by PROC1 completes as before with &1 = BEGIN, but the communication flow takes place on the request flow rather than the response flow.

In a more complex scenario, PROC1 can use this differentiation in message flows to keep separate the results of its own commands (the response flow) and the arrival of messages from operators or other processes that arrive on the request flow.

Request and Response Disciplines

It is important to understand that the terms request and response are arbitrary and have no absolute meaning. They serve only to provide a logical separation between types of messages that can be used to simplify communications between processes.

A process that issues the statement:

```
&INTREAD TYPE=RESP ARGS
```

can never receive messages sent to it by a command:

```
INTQUE TYPE=REQ
```

If you want to use INTQUE to send messages from one process to another, you must plan what you are going to send, make sure that the process you send a message to knows what to expect and is prepared to receive it, and that you coordinate the INTQUE and &INTREAD statements to make sure that they both expect the messages on the same request or response flow.

INTQUE Across ROF Sessions

Since INTQUE is a standard command, a process in one system can use INTQUE across a ROF session to talk directly to a process on the remote system. To do this, the process (or OCS operator) must know the target process identifier in the remote system and ROUTE an INTQUE command to the remote system.

This delivers an INTQUE message to the dependent response or request queue of the target process, as if the target were executing in the same system.

If the target process is executing in the remote system as a result of a ROF command, that is, if it was started in the remote system by the command:

```
ROUTE SOL2 START PROC2
```

then PROC2 cannot use INTQUE to talk back to the local system. Instead, it uses &WRITE, which in a ROF environment causes all the command output generated by a remote process to flow back to the user ID environment in the local system.

Scope of the NCL Processing Region

The concept of the NCL region is designed to limit the activities of individual users so that the NCL processes that they can run or communicate with are always within the user's own region. This prevents one user from affecting or tampering with NCL processes in use by other users or background environments.

The INTQUE command has a second authority level that is checked when a user attempts to route a message to a process outside their own NCL region. The user must have an authority level equal to or greater than the opauth value assigned to the INTQUE command. The opauth value defaults to authority level 2, but could be different in your installation.

Find Out Which NCL Processes Are Executing

Use the SHOW NCL command to display the status of active NCL processes. SHOW NCL provides displays of NCL activity by environment or by region:

- Environment shows you details of NCL processes that are executing in the NCL environment of the window from which the command is entered
- Region shows you all NCL activity associated with your user ID

SHOW NCL can also be used to display the status of NCL processes executing on behalf of other user IDs if you have sufficient command authority. For more information, see the *Reference Guide* and the Online Help.

Chapter 4: NCL Statement Types and Syntax

This section contains the following topics:

NCL Statements

An NCL procedure comprises a group of NCL statements that describe the logic and functions to be executed when the procedure is invoked. There are different types of NCL statements, where the type is determined by the function that the statement performs within the procedure.

Format of NCL Statements

NCL statements are coded as free-format, 80-character records where the first 72 characters contain the statement syntax and the last 8 characters can contain optional statement sequence numbers that, if present, are used in error messages generated by NCL.

An NCL procedure can contain statements that are completely blank. These are useful for visual layout when reading a procedure and are ignored when the procedure is executed.

Statement Continuations

Statements could require more than the 72 characters available in a single record. To accommodate this, NCL supports the continuation of a statement across multiple consecutive lines. Use of continuations can assist in improving the layout of a procedure and simplify future modification.

The number of continuations is limited to a total statement length of 2048 characters, before variable substitution.

A continuation is indicated when the last non-blank character on a statement (not including the optional sequence number in columns 73 to 80) is a plus (+) sign.

When a continuation is detected, the plus sign is removed and the text of the next statement is concatenated to the statement that contained the plus sign, after stripping leading blanks. This concatenation continues until a statement is found that does not have a plus sign as the last non-blank character.

Example 1: Statement Continuations

```
&FILE GET ID=MYFILE OPT=KEQ VARS=(A,B,+   -* DATE, TIME
                                          -* and NAME
  C,D,E,F,G,+                             -* ADDRESS
                                          -* DETAILS
  H,I,J)                                  -* EQUIPMENT
                                          -* DETAILS
```

would result in the statement:

```
&FILE GET ID=MYFILE OPT=KEQ VARS=(A,B,C,D,E,F,G,H,I,J)
```

Example 2: Statement Continuations

Continuation is deactivated by the &CONTROL NOCONT statement. This may be necessary in procedures where a plus sign is used to set a function key where the plus sign also indicates an implied blank. In such cases, these statements should be preceded by an &CONTROL statement that deactivates continuation processing and followed by another that reactivates it.

```
&CONTROL NOCONT
PF5 PREF,MSG USER1+
&CONTROL CONT
```

The CONT and NOCONT operands of the &CONTROL verb cannot be coded as variables.

Variable Substitution

Statements can contain variables that have contents that are resolved through a substitution process at the time the statement is executed. The use of variables within the statement syntax effectively changes the text of the statement by substituting the current value of the variables imbedded in the statement text each time the statement is executed.

During the substitution process, a restriction of a maximum word size of 256 characters exists. Additionally, the total size of a statement after substitution has been completed is 12288 characters.

Example: Variable Substitution

The length of the following coded statement is 28 characters, including blanks between the words:

```
&IF &A = &B &THEN &GOTO .END
```

If it were used in the following manner:

```
&A = ABCDEFG
&B = ABCDEFG
&IF &A = &B &THEN &GOTO .END
```

then, after the substitution process performed at the time the statement is executed, the statement would read as follows and be 38 characters long:

```
&IF ABCDEFG = ABCDEFG &THEN &GOTO .END
```

Variable substitution in statements let you code a standard logic routine that processes different information.

NCL Conventions and Syntax

The logic capabilities of procedures include the ability to define user and global variables, test the truth of conditions, and to make conditional branches on the basis of these tests. To achieve this, certain syntax rules must be obeyed:

- Procedure statements are stored in 80-character records. Columns 1 to 72 can be used to contain commands, statements, or comments. Columns 73 to 80 are reserved for optional sequence numbers. If present, these sequence numbers are used in error messages to pinpoint a statement in error.
- Comments can be included within procedures to assist in documentation and you are encouraged to use them freely.

  Comments can occur as separate statements or on other types of statement.
- Continuation of a statement onto the next record is supported if the last non-blank character on the line (excluding comments) is a plus sign (+). The total length of a statement including all continuations cannot exceed 2048 characters. &CONTROL CONT must be in effect.
- Syntax is free-format and can start in any column to improve presentation.
- A procedure can invoke other procedures. This process is called nesting. Nesting to 64 levels is supported. Variables can be passed to a procedure either when the procedure is invoked by the operator or when a nested procedure is invoked.
- No individual word or field within a statement can exceed 256 characters.
- The value of a variable cannot exceed 256 characters in length.

- The total length of an expanded statement after variable substitution must not be longer than 12288 characters.
- Statements can contain any number of leading or trailing blanks. A completely blank statement is ignored. No specific termination statement is required to signal the end of a procedure. When the end of the procedure is reached (end-of-file), the procedure terminates, unless terminated through other processing options (for example, &END) or errors.

Comments in NCL Procedures

NCL allows the inclusion of comments within procedures to assist in understanding the code. In addition to providing internal documentation of the processing being performed by the procedure, comments can also provide a simple means of directing messages to the operator when operating in OCS mode.

Comments fall into four categories:

- Comments on NCL statements
- Displayable stand-alone comment lines
- Highlighted key words in comment lines
- Non-displayable stand-alone comment lines

Comments on NCL Statements

Individual statements can include comments after the statement text, to describe the function being performed by that statement. In such cases the comments must commence with the two suppression characters, dash asterisk (-*). The text of the comment following the -* is free-form.

Example: Comments on NCL Statements

```
&PANEL  TESTPANEL  -* DISPLAY OUR TEST PANEL.
```

Where a statement spans multiple lines and therefore contains continuations, comments can be included on each line if the comment text follows the continuation indicator.

```
&FILE GET ID=MYFILE OPT=SEQ +-* SEARCH IS TO BE
                             -* SEQUENTIAL
           VARS=(A,B,C)      -* RECEIVE INTO THESE
                             -* VARIABLES.
```

Displayable Stand-alone Comment Lines

Comment lines can be included in procedures by inserting an asterisk (*) as the first non-blank character in the statement, unless the statement contains a label, in which case the asterisk must be the first non-blank character following the end of the label. Comment lines are displayed at an OCS window exactly as entered (both upper and lower case) after variable substitution has been performed. The leading asterisk is not displayed. This facility provides a simple means of building online operator instructions or Help facilities.

Example: Display Stand-alone Comment Lines

```
*BRING UP WESTERN SECTOR AT 13.00.
```

Highlighted Key Words in Comment Lines

Key words within a comment line can be highlighted. This is achieved by coding a plus sign (+) in place of the asterisk as the first non-blank character in the statement. Such lines are scanned for the occurrence of the highlight indicator, which is an at (@) sign. Words bounded by this character are displayed in high intensity and a blank substituted in place of the @ character. Multiple occurrences can exist within a line.

Example: Highlight Key Words in Comment Lines

```
+BRING UP@WESTERN SECTOR@AT@13.00@.
```

will display as:

```
BRING UP WESTERN SECTOR AT 13.00.
```

If only one @ is contained in a line, the remainder of that line is displayed in high intensity. The effect does not flow on to the next procedure statement.

Note: The &WRITE verb offers further facilities for displaying information and for using color and highlighting to improve information presentation.

Non-displayable Stand-alone Comment Lines

If the minus (-) suppression character (see page 50) is used in conjunction with the asterisk (*) it indicates that the comment line display is to be suppressed. In such a case the comment is not displayed, even if executing from an OCS window, and acts solely as a source of documentation within the procedure.

Example: Non-displayable Stand-alone Comment Lines

```
-*
-* THIS PROCEDURE ACTIVATES THE NETWORK
-*
```

Suppression Character

The minus character (-) is reserved as the suppression character. If used, the suppression character (-) must be the first non-blank character in the line. The suppression character can be used with both command and comment lines, but not NCL statement lines. When used in a command line, the command is executed in the normal way. However, the command is neither displayed (echoed) on your screen before execution, as would normally be the case, nor written to the system's activity log.

Example: Suppression Character

This example shows how to use the suppression character when entering commands from an OCS window.

```
-SHOW USERS
```

Results generated by the execution of the command are displayed at your terminal and are written to the activity log.

An alternative to using the suppression character is the &CONTROL NOCMD statement. This statement prevents the echoing of all commands within the procedure until the end of the procedure or until and &CONTROL CMD statement. When using NOCMD, the results generated by commands are written to the activity log.

Use of &CONTROL NOCMD has no impact on comment lines and does not cause them to be suppressed.

Label Statements

NCL supports the definition of labels that can be branched to during processing using an &GOTO or &GOSUB statement. The following rules apply when defining labels:

- Labels commence with a period (.), followed by 1 to 12 characters and not containing any ampersands (&).
- A label is not required to commence in column 1. However, it must be the first item on the statement.
- A label can be placed on a statement containing no other data.

Examples: Valid Label

```
.LABEL1        -* Normal label
.10            -* Numeric label
.ACT-NODE      -* Label containing special characters
.CMD1 statement -* NCL statement following label
```

Examples: Invalid Label

```
BAD            -* Does not commence with a period.
.LABELTOOOLONG  -* Label must be 1 to 12 characters.
.LABEL&1       -* Cannot contain ampersands.
XYZ .LABEL     -* Must be first item on line.
```

Valid Labels:

```
.LABEL1        -* Normal label
.10            -* Numeric label
.ACT-NODE      -* Label containing special characters
.CMD1 statement -* NCL statement following label
```

Invalid Labels:

```
BAD            -* Does not commence with a period.
.LABELTOOOLONG  -* Label must be 1 to 12 characters.
.LABEL&1       -* Cannot contain ampersands.
XYZ .LABEL     -* Must be first item on line.
```

Label Variables

A variable can be specified as the target of an &GOTO or &GOSUB statement. This variable, after variable substitution, is then taken as a label to which a branch is required.

Example: Label Variables

```
&GOTO .&MSGID
```

can read, after variable substitution:

```
&GOTO .IST350I
```

and a search for the label .IST350I is made.

Usually, most information input to a procedure contains a unique identifier such as a message number. The ability to branch to a label variable makes it possible to use this unique message number and branch directly to the part of the procedure concerned with processing the message, instead of having to use a number of &IF statements to determine the processing required.

Undefined Labels

Normally, an attempt to branch to an undefined label results in an error and the procedure is terminated. However, the &CONTROL NOLABEL operand lets the user specify that an attempt to branch to an undefined label will not result in an error, but will simply drop through and execute the statement following the &GOTO. Thus, a trap for unexpected messages can be set up and all of the efficiencies associated with direct branching still achieved.

Example: Undefined Labels

```
&CONTROL NOLABEL
.NEXT
     &PPOREAD VARS=(A,B,C,D,E)
     &GOTO .&A
     &WRITE ALARM=YES DATA=UNEXPECTED INPUT &A &B &C &D &E
     &GOTO .NEXT
.IST305I
.IST314I
     .
     .
     .
```

Duplicate Labels

A label in an NCL procedure should not appear more than once; NCL normally checks to ensure a label is not duplicated. If the &CONTROL NODUPCHK operand is used, duplicate label checking is not performed. This option can result in improved efficiency, but is normally only used after the procedure has been thoroughly debugged.

While label variables offer great benefits in both performance and simplicity, one point needs to be considered. By its very nature, a label must be unique within a given procedure. An attempt to &GOTO a duplicate label will result in an error (unless the &CONTROL NODUPCHK option is in effect). If the processing of a label variable yields a duplicate label, perhaps conflicting with one used earlier in the procedure, you can use the following. if you are using a variable in an &GOTO that is itself not unique, you can append one or more characters to the label in the &GOTO to ensure the generated label is in fact unique.

Example: Duplicate Labels

In this example, the operator can enter YES or NO to either of the &PAUSE statements. In the second, the letter X is prefixed to the operator's input to differentiate it from input entered in reply to the first &PAUSE statement.

```
&WRITE DATA=ENTER 'Yes' OR 'No'
.PAUSE &PAUSE ARGS
     &GOTO .&1
     &WRITE DATA=INVALID RESPONSE, RE-ENTER.
     &GOTO .PAUSE
     .NO &END
     .YES
     .
     .
     .
     &WRITE DATA=ENTER 'Yes' OR 'No'
.PAUSE2 &PAUSE ARGS
&GOTO .X&1
&WRITE DATA=INVALID RESPONSE, RE-ENTER.
&GOTO .PAUSE2          -* Note addition of X to both
.XNO &END              -* &GOTO and target labels to
.XYES                  -* make labels unique.
```

Minimize Labels

By using the verbs &DO, &DOWHILE and &DOUNTIL in your procedure logic you reduce the need for &GOTO and &GOSUB statements. This is recommended, both to reduce the number of labels that you need in a procedure and to improve the structure of your procedures.

Verb Statements

NCL verbs cause actions to occur. There are different types of verbs, some that dictate the flow of processing and logic, others that fetch information for the procedure to process and others that cause data to flow to external targets.

The following are examples of verb statements:

&DOWHILE

Causes processing to continue in a loop while a certain condition exists.

&WRITE

Allows a procedure to write a message to the user's terminal.

&FILE

Allows a procedure to read, write, or delete records on a file.

NCL statements that contain a verb have the following general form:

```
verb [ opt=opt  &variable ...  &variable ]
```

where verb is the keyword that is the name of the verb to be executed, opt is a keyword operand identifying a sub function of the verb, and &variable represents parameters required as input to the verb's actions, or the names of variables that receive the results of the verb's execution.

Built-in Function Statements

NCL built-in functions perform commonly required functions that are either impossible to achieve using NCL or require complex NCL coding.

The following are examples of built-in functions:

&DEC

Converts a hexadecimal number to its decimal equivalent.

&SUBSTR

Lets you extract part of one variable and place it in a second variable.

&CONCAT

Lets you concatenate the values of two or more source variables and place the result in a target variable.

NCL statements that contain a built-in function have the general form:

```
&target = function  parameter
```

where &target is the name of a variable that is to receive the result of the built-in function, function is the keyword that is the name of the built-in function to be executed and parameter represents parameters required as input to the built-in function.

Note: For more information about NCL verbs and built-in functions, see the *Network Control Language Reference Guide*.

Assignment Statements

Assignment statements let you set a new value for a nominated variable. Assignment statements have the following format:

```
&variable = expression
```

where &variable is the name of the target variable, whose value is to be changed to the result of the execution of expression. The target variable is always on the left of the equals sign (=) and the value to be assigned to the target is always on the right of the = sign.

Assignment statements are used to manipulate the values of the variables that your procedure uses.

Assignment also occurs as the result of executing built-in functions and some verbs.

Example: Assignment Statements

```
&A = 10                      -* &A is assigned the value 10
&XYZ = ABCDEF                -* &XYZ receives the value ABCDEF
&A = (&C + 1) - (&B * 5)     -* Add 1 to &C then subtract
                             -* five times the value of
                             -* &B and put the result in &A.
```

Arithmetic

Arithmetic is performed by a series of explicit assignment statements, in which the target variable receives the result of the calculation coded to the right of the = sign. Compound arithmetic expressions are supported by NCL, and both integer and floating-point arithmetic can be used.

More information:

About Arithmetic in NCL (see page 83)

Command Statements

You can execute any product command within a procedure by coding the command statement exactly as you would enter the command from an OCS window.

You can also code the command statement to include variables in the command text, in which case substitution is performed before the command is executed.

Chapter 5: Variables, Substitution, and Assignment

This section contains the following topics:

What Is a Variable?

The term variable describes a word that may represent different values within a statement. A variable commences with an ampersand (&) and is followed by 1 to 12 characters that form the name of the variable.

A variable name can contain the characters $, #, @, A to Z, and 0 to 9. The name is delimited by the first blank or non-valid character. Although the names of variables can be coded to contain both upper and lower case characters (A to Z) they are treated internally as being entirely upper case.

However, when operating in a system where support for double byte character stream (DBCS) terminals is active, no translation to upper case is performed.

If a variable name starts with a digit, the whole variable name must be numeric.

Variable Types

There are four types of variables:

- System variables
- User variables
- Global variables
- Parameters

System Variables

A number of variables are maintained by the system to provide access to commonly required facilities. An example of a system variable is the time, which can be obtained by referencing the &TIME system variable.

User Variables

Any number of user variables (within storage limitations) can be defined to contain information required during the processing phase of a procedure. Explicit definition of user variables is not required. The first time a user variable is referenced it is automatically defined. User variables are automatically deleted when a procedure terminates.

User variables can be passed across nested procedure levels if the &CONTROL SHRVARS operand is in force when the nested procedure is invoked, otherwise they are unique within the current nesting level. If a user variable is required in a nested level and &CONTROL SHRVARS is not in effect, its value must be passed explicitly as a parameter on the EXEC command when the nested level is invoked. An &RETURN statement can then be used to return specified variables to a higher nesting level, thus facilitating the development of modular functions.

Global Variables

Global variables operate in a similar manner to user variables. The difference is that global variables, as the name implies, are global to the entire system and once created can be referenced or modified by any procedure. Thus, global variables can be used to perform cross-talk between procedures.

Global variables commence with a 1- to 4-character prefix that you can set for your site using the SYSPARMS NCLGLBL command. Unless otherwise defined this prefix defaults to GLBL. The remaining characters must conform to standard variable naming conventions.

The &000 system variable is set to the current value of the global variable prefix. Thus, if the SYSPARMS NCLGLBL=#@ command was used to set the global variable prefix to #@ then &000 returns that value. This facility is provided to allow the procedure designer to develop procedures that execute correctly regardless of the global variable prefix. For example, &&000SYS1 is resolved to &GLBLSYS1 using the default value for the global variable prefix, or &#@SYS1 if the prefix is set as mentioned previously.

A global variable can be created explicitly by an assignment statement, for example:

```
&&000SHIFTLDR = &STR SHIFT LEADER IS BILL SMITH
```

It can also be created implicitly by specifying a global variable name as the target of a verb that creates or modifies variables as part of its function.

Once created, a global variable remains in existence until it is explicitly deleted by an assignment statement. For example:

```
&&000SHIFTLDR =
```

Note: The uncontrolled creation of large numbers of global variables can consume a large amount of storage. You should ensure that global variables are deleted as soon as they are no longer needed. Care should be taken when designing procedures to ensure that they do not end without deleting any redundant global variables created.

CA recommends that you observe a standard naming convention for global variables to avoid the inadvertent destruction of variables where, for example, one procedure creates a global variable with an identical name as that created by another.

The SHOW NCLGLBL command provides a display of the global variables that exist in the system. For more information, see the Online Help.

Persistent Global Variables

You can save selected NCL global variables to preserve data between restarts of the region. Saved global variables are known as persistent global variables (PGVs). They are automatically loaded at restart.

You can use the GLBLSAVE process macro or calls to the Persistent Global Variables Interface ($CAGLBL) to create PGVs. GLBLSAVE has an NCL procedure ($RMMC47S in the COMMANDS data definition) that shows how to use $CAGLBL.

Note: For more information about $CAGLBL, see the *Network Control Language Reference Guide*.

You can administer PGVs from the Persistent Global Variables List by entering /PVAR at the command prompt.

Note: For more information about the Persistent Global Variables List, see the Online Help.

Parameters

Parameters are those variables passed to the procedure when it is invoked by the operator, by another procedure, or by your product.

Parameters entered when the procedure is invoked are positional and must be entered in the order expected by the procedure. Entered parameters are separated by one or more blanks. Commas are not accepted as delimiters between parameters. The procedure must verify that the entered parameters are correct. Each parameter is allocated to a variable in the form &n, where n is a number, starting at 1, identifying the position of the parameter.

If the operator enters:

```
EXEC TEST1 PU1 LU2 NCP3
```

the following variables are created:

```
&1  will be set to PU1
&2  will be set to LU2
&3  will be set to NCP3
```

In addition, when a procedure is invoked, the user variable &ALLPARMS will be set to the entire string provided, excluding the EXEC or START command and the procedure name. In the previous example, &ALLPARMS is set to:

```
PU1 LU2 NCP3
```

The system also supplies a count of the number of variables created in the system variable &PARMCNT. In the previous example, &PARMCNT is set to 3.

Note: &CONTROL SHRVARS can be utilized to pass some or all variables to a lower nesting level rather than specifying individual parameters.

DBCS Device Support

Your products support terminals that use DBCS representation of symbols for languages such as Japanese (Kanji script).

When DBCS support is active, the system no longer automatically translates lower case characters to upper case for comparative purposes, and so on. When DBCS is active all characters in the extended character set are regarded as unique, and certain system functions no longer apply, or are treated as a no-operation.

Variable Substitution

Before a procedure statement is analyzed or a command is executed, the record is scanned for the presence of variables, indicated by an ampersand (&). This process is called variable substitution. Variable substitution is performed from right to left of the statement. When a potential variable is detected by the presence of an ampersand, the variable is isolated by scanning to the right for a delimiting blank, comma, or other character not valid in a variable name.

Note: A single character ampersand is not treated as a variable and remains intact as a single ampersand.

Take the string &A.1 as an example. &A is processed as the variable and the resulting substitution inserted in its place (for example, 123.1).

Variables whose names are entirely numeric (for example, &99) have unique properties in that they are delimited by the first non-numeric character. For example:

```
&91 = 123

&A  = &91ABC
```

The variable &A in this example would yield the value 123ABC. This technique can provide an alternative to the use of &CONCAT with alphanumeric variables.

In the substitution process no spaces are added or deleted. The variable is removed and the text to the right of the variable is relocated to accommodate the data being substituted in place of the variable. A variable can be referenced as often as required. Left and right alignment of substituted data can be achieved using the &CONTROL ALIGNL and ALIGNR options to enhance tabular output or when displaying data in full-screen panels the #FLD statement can be used to assign specific alignment attributes to a field.

Undefined Variable Substitution

Undefined variables, or variables that have a null value, are eliminated from the line; therefore, you should to ensure that variables are correctly coded and will not be eliminated during the substitution process. For example, the following statement:

```
&WRITE DATA=The time is &TIMW and date is &DATE1
```

incorrectly contains the variable &TIMW where &TIME was intended. Because &TIMW does not have an assigned value, it is eliminated from the statement; the final result written to the user's terminal is:

```
The time is  and date is 91.001
```

In certain cases, such as with an &IF statement, this elimination of variables can pose problems and result in syntax errors. Consider the following statement:

```
&IF &1 EQ YES &THEN &GOTO .OK
```

If &1 is a null value, perhaps because an operator did not enter its value after an &PAUSE, the statement after substitution appears as:

```
&IF EQ YES &THEN &GOTO .OK
```

and results in a syntax error and the procedure terminates.

A technique that can be used to avoid this is to append a constant character to the variable and, of course, to the value to which it is to be compared. For example:

```
&IF .&1 EQ .YES &THEN &GOTO .OK
```

In this case, if the variable &1 has a null value, the following statement results:

```
&IF .  EQ .YES &THEN &GOTO .OK
```

which is syntactically correct and performs as desired.

Note: When comparing numeric values, a zero (0) can be used as the constant character.

For more information about using this technique, see the &IF verb description in the *Network Control Language Reference Guide*.

Complex Variable Substitution

Complex or multi-level variables are supported. If after one substitution, the value generated remains a variable (commences with an &), substitution is again performed.

Example 1: Complex Variable Substitution

```
&A = 1
&1 = MESSAGE
&WRITE DATA=TEST &&A
```

Because the current value of &A (1) is substituted into the statement, the first pass of the &WRITE statement yields:

```
&WRITE DATA=TEST &1
```

The second pass then resolves &1, and yields:

```
&WRITE DATA=TEST MESSAGE
```

Example 2: Complex Variable Substitution

Complex variables are important when performing matrix processing where an unknown number of variables must be dynamically generated to accommodate data.

```
&FILE OPEN ID=MYFILE
&FILE SET ID=MYFILE KEY=MYKEY
&LIMIT = 100
&CNT = 1
&DOWHILE &CNT LT &LIMIT
         &FILE GET ID=MYFILE OPT=KEQ VARS=MYFIELD1
         &SAVE&CNT = &MYFIELD1
         &CNT = &CNT + 1
&DOEND
```

Align Substitution Data

The substitution process does not, by default, preserve any difference in the length between a variable name and the data being substituted in place of the variable. This can make tabular output of rows of numbers difficult or impossible to align when not using full-screen facilities.

The &CONTROL statement provides the ALIGNL and ALIGNR options to specify special alignment requirements. ALIGNL requests left alignment and ALIGNR right alignment. Both of these options can additionally identify a fill character (by default a blank) that is to be used, for example, ALIGNR$.

When using these alignment options, the length of the name of the variable itself is used to determine both the point of alignment and the number of fill characters required. Therefore variables with names of the same length must be used to ensure the correct alignment is achieved.

Example: Aligning Substitution Data

In this example, the three variables &COUNT01, &COUNT02 and &COUNT03 are used to display a table of numbers. They are to be right-aligned and padded with leading dashes (-).

```
&CONTROL ALIGNR-
&COUNT01 = 1
&COUNT02 = 1098
&COUNT03 = 66
&WRITE DATA=COUNT 1 = &COUNT01
&WRITE DATA=COUNT 2 = &COUNT02
&WRITE DATA=COUNT 3 = &COUNT03
```

The result is:

```
COUNT 1 = -------1
COUNT 2 = ----1098
COUNT 3 = ------66
```

Note: These techniques are designed to be used by NCL procedures running in the OCS environment which do not utilize full-screen panels.

Lowercase Data

By default, data assigned to a variable is converted to uppercase. However, use of the &CONTROL NOUCASE statement allows lowercase data, such as input from a full-screen panel, to be maintained. If you want comparison operations to occur without translation of the operands being compared, specify &CONTROL NOIFCASE in addition to the NOUCASE option.

Note: For data entry into a panel in lowercase, the field definition for the panel must also specify CAPS=NO.

When operating in a system with the SYSPARMS DBCS=YES option specified, no distinction is made between uppercase and lowercase. The rules regarding automatic conversion to uppercase do not apply.

Debugging Procedures

During the development of procedures coding errors are bound to occur. The &CONTROL TRACE and &CONTROL TRACELOG statements allow you to request statements to be written to your terminal or the system log after variable substitution and before execution. The &CONTROL statement can occur as frequently as required and can therefore be placed strategically within the procedure.

The TRACELAB and TRACEALL options are available to provide label flow tracing and to include additional detail on the trace records that are logged.

The &CONTROL NOTRACE statement returns to normal processing. You will have imposed a limit to the maximum number of trace messages that can be generated by a procedure. By default this limit is 100 messages and has been imposed to stop excessive tracing consuming system resources.

The NCLTRACE command can be used from an OCS window to dynamically start or stop NCL procedure tracing while an NCL procedure is executing. This lets you debug executing processes without having to stop them, edit the NCL to include &CONTROL statements, and then restart them.

For more information about the NCLTRACE command, see the Online Help. For more information about the &CONTROL verb, see the *Network Control Language Reference Guide*.

When testing procedures that use full-screen processing, the temporary inclusion of variables in a vacant area in a panel can assist in tracking and development. These can be removed when the procedure has been completed.

The &WRITE statement can also be used to great advantage to display the contents of variables at specific points in processing.

When using &WRITE from procedures operating in full-screen mode, the messages are queued and displayed in one or more full-screen panels when the procedure terminates. Having completed the display of any queued messages, the procedure ends.

Set Variables to a Particular Value

Assignment is the term used for the process of setting variables to a particular value. Assignment is either explicit, in which case you use the assignment statement to change the value of a variable, or implicit, in which case the action of a verb causes a change in value of a target variable (or variables).

Explicit Assignment: Assignment Statement

The basic format of the assignment statement is:

```
&variable = expression
```

&variable

Is the name of the user variable or global variable whose value is to be changed. The variable might not actually exist, in which case the assignment statement creates it then sets its value.

The name of the variable must be valid according to the rules for variable names.

If a system variable is specified as the target of an assignment statement a syntax error will occur.

Complex variables (see page 67) are supported.

=

Is written as shown, indicating that an assignment is required.

expression

Represents the value that is to be assigned to the target variable. If no expression is supplied the target variable is assigned a null value, and the variable is automatically deleted. An expression is one of the following constructs:

- A variable or constant whose value is to be assigned to the target variable.
- A built-in function such as &CONCAT, &SUBSTR, and so on. When a built-in function executes it produces a result that is used as the value to be assigned to the target variable.
- An arithmetic expression. The arithmetic calculation is performed to yield a result, which is then used as the value to be assigned to the target variable.

Complex Variables in Assignment Statements

Use of complex variables within an assignment is supported. A complex variable requires multiple substitutions to determine the final variable required. For example:

```
&&1
&A&1
```

An example of the use of complex variables is the accumulation of counts for different values. A counter is initialized for each possible value. A single statement can then be used to increment the associated counter.

Example: Complex Variables in Assignment Statements

In this example, a variable (&1) can have the values A, B, or C and you want to count the occurrences of each of these. This could be achieved in the following way:

```
&CNTA = 0
&CNTB = 0
&CNTC = 0
   .
   .
   .
&DOWHILE &CNT&1 NE &LIMIT
   .
   .
   processing to receive input and set &1 to A, B or C
   .
   .
   &CNT&1 = &CNT&1 + 1
&DOEND
```

The assignment statement target (&CNT&1) will be resolved to &CNTA, &CNTB or &CNTC prior to execution.

Implicit Assignment: Using &ASSIGN

The &ASSIGN statement, which is an NCL verb, is used to assign values to multiple target variables in one operation. It is common for procedures to operate with groups of related variables and to have the need to manipulate the values of all the variables in a group.

Using explicit assignment statements to change the values of all the variables in a group is less efficient than using the &ASSIGN statement.

&ASSIGN lets you define named or generic groups of source variables, and transfer source variable contents to corresponding target variables, also named specifically or generically. Similarly, groups of target variables can be assigned null values and deleted in a single operation.

Implicit Assignment: Using Other NCL Verbs

While &ASSIGN is a verb designed specifically to allow multiple assignment operations, many other NCL verbs also cause assignment of a new value to one or more target variables as the result of their execution. Examples of these NCL verbs are &PARSE and &SETVARS. A complete description of these verbs and how to use them can be found in the *Network Control Language Reference Guide*.

Verbs whose action results in retrieving information from an external source (for example, &FILE, &PAUSE) place the data they receive into specified target variables. An implicit assignment of values takes place as a result of the execution of this type of verb.

Uppercase and Lowercase Variables

By default, when character data is assigned to a target variable, the data is converted to uppercase.

If you want to manipulate lowercase character data, use the &CONTROL NOUCASE processing option within your procedure. The NOUCASE option stops the automatic translation of character data to uppercase on assignment statements.

DBCS Support and Lowercase Data

If your system is operating with DBCS support active (SYSPARMS DBCS=YES), then the system regards all data as a single case, and no uppercase or lowercase translation is performed. In this execution mode, the &CONTROL UCASE option is ignored.

Variables and Storage Usage

Variables hold the data your procedures work with. This data occupies virtual storage in your product region or partition. Therefore, the more variables you create the more storage your procedures will use.

While your product attempts to optimize storage usage, you should design your procedures to control the use of variables and avoid creating large numbers of variables that are seldom referenced.

If the temporary use of an extremely large number of variables is required, then assign a null value to these variables when they are no longer required to delete them and reduce the storage required by the procedure.

The &ASSIGN statement can be used to nullify large groups of variables in a single operation.

All storage used for NCL variables resides above the 16 MB line.

NCL Table Manipulation

The discussion of variables and substitution has so far concentrated on using simple and complex variables as single data entities. You can also construct tables of data using multiple individual variables using either of two common techniques:

- You can use complex variables, for example, &DATA&INDEX where the variable &INDEX contains a number (for example, 1,2,3), so generating variable names of &DATA1, &DATA2, &DATA3, and so on.

  This technique is fairly efficient for small tables, but the overheads of creating and managing large numbers of variables become significant when dealing with large tables. However, it is only suitable for numeric table indexes, or short alphanumeric indexes that do not contain any characters that are invalid in an NCL variable name.

- The second technique uses the complex variable approach described previously, but each entry in the table is represented by 2 (or more) variables (for example, &KEY&INDEX, and &DATA&INDEX). One contains the key value, and the others the data values. The table is searched by incrementing an index variable, and comparing the key variable to the search value. When a match is obtained, the index variable is used to build the complex variable name containing the data.

Vartable Facility

A vartable is an in-storage table with the following attributes:

- A name, assigned when the table is created, and used to refer to the table.
- A scope, which can be one of:

  PROCESS

  (Default) Indicates that the table is only visible to the NCL PROCESS that allocated it. It is automatically freed when the process terminates, if not explicitly freed beforehand. Table names must be unique within a process, but different processes can operate using different tables with the same name.

  REGION

  Indicates that the table is visible to all NCL processes in a processing region. For a signed-on user, this means all NCL processes running in either NCL window, under the User Services menu, MAI script procedures, and so on. The table is automatically freed when the region terminates (for example, when the user signs off), if not explicitly freed beforehand. Table names must be unique within the region, but different regions can use different tables of the same name.

  SYSTEM

  Indicates that the table is visible to all NCL processes in this address space. The table is automatically deleted when your product terminates, if not explicitly freed sometime beforehand.

  SYSTEM scope replaced GLOBAL in an earlier version but the use of GLOBAL is supported.

  AOM

  Indicates that the statement refers to a mirrored vartable (see page 78) if AOM has started.

- A key length, from 1 to 256 characters long. Each table can have a different key length.

- Zero or more entries, consisting of:
  - A key. Every entry must have a unique key, which can be numeric.
  - Up to 16 items of data, or an unlimited number of data items if DATA=MAPPED is specified. Each item can be from 0 (null, not present) to 256 characters long. The data value is null if no data is provided for an item when an entry is added.
  - A counter field, which is initialized to 0 when an entry is created, and can be reset to any value, or adjusted by any amount, on update calls.
  - A user correlator field. This field is maintained by the system, and can be used to provide update synchronization for tables shared between NCL processes (that is, for tables with SCOPE= REGION or SYSTEM only).

The entries in a table are maintained in ascending key field value order. The retrieval option of &VARTABLE allows sequential retrieval in ascending or descending order.

&ZFDBK Values

The &ZFDBK system variable is set by the &VARTABLE verb to indicate the success of the action. Most of the possible values are documented in the *Network Control Language Reference Guide*. Vartables with SCOPE=AOM can also return other &ZFDBK values. These are fully described under each &VARTABLE option in the *Network Control Language Reference Guide*.

&VARTABLE Manipulation Facilities

&VARTABLE provides the following table manipulation facilities:

ADD

Add an entry to a vartable.

ALLOC

Allocate a new vartable.

DELETE

Delete an entry from a vartable.

FREE

Free (unallocate) an existing vartable.

GET

Get (retrieve) an entry from a vartable.

PUT

Add or update (if there) an entry in a vartable.

QUERY

Query the existence of a vartable, and optionally return attribute information.

RESET

Delete all entries in an existing vartable, but preserve the definition of the table (like doing FREE/ALLOC).

UPDATE

Update an existing entry in a vartable.

The functions ALLOC, FREE, RESET, QUERY provide table-level manipulation. ADD, PUT, UPDATE, DELETE, GET are used to manipulate entries in a table.

Shared Table Updating

Tables allocated with a scope of REGION or SYSTEM can be accessed by any number of concurrently running NCL procedures. An NCL procedure can be interrupted at any time and another procedure can be scheduled. If two procedures are manipulating the same table, the possibility of logical corruption exists.

Example: Shared Table Updating

In this example, assume that the entry with key KEY001 has a data content of 10 before the code is executed.

```
procedure 1 procedure 2

&K = KEY001
&VARTABLE GET ID=T1 +
            SCOPE=SYSTEM +
            KEY=K FIELDS=D VARS=DATA
&D = &D + 1  -* &D now 11
-*
-* system interrupts procedure 1, schedules procedure 2
-*

                     &K = KEY001
                     &VARTABLE GET ID=T1 +
                                 SCOPE=SYSTEM +
                                 KEY=K FIELDS=D VARS=DATA
                     &D = &D + 1  -* &D now 11
                     &VARTABLE PUT ID=T1 +
                                 SCOPE=SYSTEM +
                                 KEY=K FIELDS=D VARS=DATA
                     &END
-*
-* system re-schedules procedure 1
-*
&VARTABLE PUT ID=T1 +
            SCOPE=SYSTEM +
            KEY=K FIELDS=D VARS=DATA
```

If the data in the table entry was being used as a counter, instead of having 12 (because of 2 adds), it only has 11, as the update done by procedure 2 is lost.

There are three solutions to the problem:

- If a single counter is being maintained in each entry, there is no need to get, alter, then update the table entries. Instead, you can use the ADJUST= or FIELDS=(ADJUST) options of the &VARTABLE PUT, UPDATE, or ADD operands to change the system-maintained counter fields in the entry without interference from any other procedure that is using the table. The previous example could be written as:

  ```
  &K = KEY001
  &VARTABLE PUT ID=T1 SCOPE=SYSTEM +

             KEY=K ADJUST=1
  ```

  This would insert a new entry if it did not already exist, and add 1 to the (zeroed) counter, or update an existing entry by adding 1 to the current counter value.

  This approach is extremely useful for event counting, where the event (for example, a particular message ID) is described by the key.

- Use the &LOCK verb to lock the key value during the manipulation:

  ```
  &LOCK TYPE=EXCL .....  use key as resource name....
  &VARTABLE GET ...
  ...  manipulate the table entry
  &VARTABLE UPDATE ...
  &LOCK TYPE=FREE .....
  ```

  The &LOCK verb is described in the *Network Control Language Reference Guide*.

  This approach allows almost any manipulation to take place. However, for heavily accessed tables, the overheads of LOCK/UNLOCK could be excessive. If you do not know the key of an entry in advance, you might need to lock the entire table.

- If we assume that minimal contention for any one table entry, a third approach is to consider we can access it and only redo our work if someone else actually changes the entry while we are performing calculations.

  Remember that each entry in a vartable contains a user correlator. This is a system-maintained update counter. An &VARTABLE GET operation can request that the current value of the correlator for the requested entry be returned in a nominated variable. The actual format of the correlator is of no consequence, and should not be altered by the procedure.

  When the procedure issues the &VARTABLE UPDATE, &VARTABLE PUT, or &VARTABLE DELETE, the name of the variable containing the correlator should be supplied in the VARS list (and .USERCORR specified in the FIELDS list). The system checks that the correlator matches the current value of the correlator in the current table entry, and only if they are the same, performs the update or delete operation (if PUT, and the entry is new, the correlator is ignored).

  Following the update, the correlator in the table entry is given a new, unique value (again, of no consequence to the procedure).

  Obviously, if two procedures issue a GET for the same entry, with no intervening updates to that entry, they will both be given the same correlator value. But, if both then attempt to update that entry, supplying the same correlator value, the second update will fail (with &ZFDBK set to 8), and it should reissue the GET, and redo the update logic.

Example: Shared Table

```
&K = KEY001
.RETRY     -* loop here if update fails on correlator
           -* mismatch

&VARTABLE GET ID=T1 SCOPE=SYSTEM KEY=K +
        FIELDS=(.DATA,.USERCORR) VARS=(D,UC)
           -*
           -* manipulate the entry data (in &D).
           -*
&VARTABLE PUT ID=T1 SCOPE=SYSTEM KEY=K +
        FIELDS=(.DATA,.USERCORR) VARS=(D,UC)

&IF &ZFDBK = 8 &THEN &GOTO .RETRY-* loop if
                                 -* changed
```

Note: Because no locks are held, there is no extra work to release those locks should the procedure encounter an error condition while manipulating the data.

Also, if the GET is using an OPT= that retrieves a different key value record (for example OPT=GEN), you do not need to know in advance what key value to lock. If &LOCK had to be used in this case, you might need to lock the entire table.

Note: The use of the user correlator can be forced, for a particular table, by specifying USERCORR=YES when allocating it.

(To contrast these two views, (2) can be regarded as the conservative approach, and (3) can be regarded as the aggressive approach.)

Retrieval Techniques

The &VARTABLE GET statement allows an NCL procedure to retrieve an entry from a vartable, placing the key, data, counter, and possibly user correlator, into nominated NCL variables. The specific entry returned depends on both of the following:

- The value of the OPT= parameter on the &VARTABLE GET statement
- The value contained in the NCL variable nominated on the KEY= parameter of the &VARTABLE GET statement, unless OPT=FIRST or OPT=LAST is specified, in which case the KEY= parameter is not permitted.

The default OPT= parameter is OPT=KEQ, which searches the nominated vartable for an exact EQUAL KEY match.

Most of the other OPT= values are straightforward:

KGE

Retrieve the entry with the lowest key value greater than or equal to the supplied key value.

KLE

Retrieve the entry with the highest key value less than or equal to the supplied key value.

KGT

Retrieve the entry with the lowest key value greater than the supplied key value.

KLT

Retrieve the entry with the highest key value less than the supplied key value.

GEN

Retrieve the entry with the lowest key value generically equal to the supplied key value (that is, match for as many non-blank characters in the supplied argument, and the fewest non-blank characters after).

Two OPT= values allow retrieval of the entry with the lowest key (OPT=FIRST), or highest key (OPT=LAST). When used in conjunction with the OPT=KGT or OPT=KLT options in a loop, a table can be sequentially read (the &VARTABLE GET description illustrates this).

OPT=IGEN

The remaining OPT= value, OPT=IGEN (Inverse GENeric), has special use. The definition is: Retrieve the entry with the longest key value equal to the supplied key value, but at least one character long. For example, a search key of ABCDE searches as follows:

1. For ABCDE; if no match is found, the search continues for...
2. ABCD; if no match, it would then search for...
3. ABC; and so on through...
4. AB and...
5. A

This retrieval option is especially suited for screening messages by message identifier, where generic identifiers (for example, IEF) control all messages that start with that identifier, and less-generic identifiers (degenerating to specifics, such as IEF431I) override the generic identifier in a particular case.

By loading a table with these identifiers, an incoming message identifier can be looked up in one GET OPT=IGEN statement, to find the most-specific match. This avoids looping to examine lists of shorter and shorter identifiers.

&VARTABLE Syntax Descriptions

For a complete description of the &VARTABLE verb and its options, see the *Network Control Language Reference Guide*.

Mirrored Vartables

Note: Mirrored vartables are only available if your region includes Automation Services products.

A mirrored vartable can be used to control an AOM screening table. For example, message IDs can be stored in a mirrored vartable, and message suppression and routing can be controlled dynamically by adding or deleting entries in the table. Using the LOOKUP screening statement, messages that are matched by message ID can have various attributes assigned or altered.

The ability to dynamically add entries to a mirrored vartable, by screening table code, allows statistics on such things as message ID to be easily accumulated.

If AOM is not started when a vartable with a scope of AOM is allocated, no mirroring takes place. When AOM START is issued, all existing SCOPE=AOM vartables are mirrored. When AOM STOP is issued, all mirrored vartable copies are deleted; the standard vartable remains. If a mirrored vartable is allocated while AOM is started, it is mirrored immediately.

Differences Between Mirrored and Standard Vartables

A mirrored vartable is a standard vartable, allocated with a scope of AOM. SCOPE=AOM is similar to SCOPE=SYSTEM; any NCL procedure can refer to a SCOPE=AOM vartable, but these vartables are logically separate to SCOPE=SYSTEM. That is, there can be a table with an ID=TAB1 in both SCOPE=AOM and SCOPE=SYSTEM.

Mirrored vartables are distinguished from standard vartables by using the &VARTABLE verb to maintain mirrored vartables. This copy can be read from, added to, and updated by an AOM screening table.

In z/OS, the mirrored copy is maintained in (E)CSA; in VM, it is maintained in your product storage. SYSPARMS AOMMIRST=n sets the maximum amount of storage in kilobytes that can be used by mirrored vartables.

Update Mirrored Vartables

When a new entry is added to the mirrored vartable by an NCL procedure, the new entry is automatically copied into the mirrored copy of the table.

The mirrored copy contains the following:

- A copy of the key (always 16 characters)
- A copy of the first 8 bytes of the DATA1 data field (blank padded if necessary)
- The .AOMID attribute
- An encoded version of the .AOMATTR attribute string
- A counter field .AOMCOUNT

When an existing entry is updated or deleted, the appropriate action is also performed on the mirrored copy. The screening table LOOKUP statement cannot update data in an entry, or delete an entry, but can update the count field in the mirrored entry.

When a screening table LOOKUP statement adds a new entry, the downward mirroring is not performed immediately, but is performed as soon as is necessary. This happens when:

- &VARTABLE GET retrieves an entry with that key value.
- The entry can be sequentially accessed.
- GET OPT=FIRST or LAST is specified. In this instance all pending mirroring is performed immediately.

These rules allow the mirrored copy of the vartable to be accessed with no serialization. This enhances the performance of the screening table.

AOM Attributes of Mirrored Vartables

Mirrored vartables support several extra data fields in a table entry.

.AOMID

This attribute allows a 1- to 12-character AOM identifier to be stored or retrieved. When a LOOKUP statement assigns the ID attribute on a successful lookup, this value is assigned as the ID of the current message and is available to AOMPROC in the &AOMID system variable. If the supplied value is blank, the AOMID in the table entry is regarded as not specified, and a LOOKUP ASSIGN statement of ID will not override the current ID value for the message.

.AOMATTR

This attribute lets you set a list of AOM message attributes. Each allowable attribute is encoded into a 1- or 2-character value and placed into a particular position in a character string. The blank and dash (-) have special meaning. A blank in any position means that the associated attribute is to be regarded as not specified. This means that when a LOOKUP statement assigns an attribute from a table entry that is not specified, the current attribute value is not changed. A dash (-) means that the current value of the associated attribute is not to be changed and acts as a place holder when updating attributes that are further down in the attribute string.

The following message attributes can be encoded in a .AOMATTR string:

Position 1

z/OS delete option. Y-YES, N-NO, F-FORCE.

Position 2

ALARM option. Y-YES, N-NO.

Position 3

MONITOR option. Y-YES, N-NO.

Position 4

INTENSITY option

H-HIGH, N-NORMAL, L-LOW.

Position 5

NRD option. Y-YES, N-NO, O-OPER.

Position 6

TRACE option. S-START, F-FINISH, *-This one.

Position 7

ROUTE option. P-PROC, O-PROCONLY, M-MSG, B-BOTH, L-LOG, N-NO.

Refer to local/remote route options. If LCLROUTE and/or RMTROUTE are also specified, they override any value set here. If both positions are set and are different, this position is returned as not specified (-).

Position 8

HLITE option. N-NO, R-REVERSE, B-BLINK, U-UNDERSCORE,

Position 9

COLOR option. N-NO, B-BLUE, R-RED, P-PINK, G-GREEN, Y-YELLOW, T-TURQUOISE, W-WHITE.

Positions 10-11

MSGCODE value. 2 hexadecimal digits.

Positions 12-19

8 user flags. Each can be Y-YES, N-NO.

Position 20

LCLROUTE option. P-PROC, O-PROCONLY, M-MSG, B-BOTH, L-LOG, N-NO

Position 21

RMTROUTE option. P-PROC, O-PROCONLY, M-MSG, B-BOTH, L-LOG, N-NO

Position 22

DOM-TRACK option. Y-YES, N-NO.

Positions 23-30

8 RMTCLASS values. Each can be Y-YES, N-NO.

If a string shorter than 30 characters is specified it is padded to 30 characters with dashes. A new entry is regarded as having all attributes as not specified prior to processing any supplied .AOMID or .AOMATTR values.

When retrieving a table entry, if the .AOMATTR values are retrieved, a 30-character string is always returned, however some positions can be blank.

.AOMCOUNT

This attribute cannot be set or updated but can be retrieved by using the &VARTABLE GET statement. This is a counter that is incremented if the COUNT option is specified in the LOOKUP statement. It can be used as a hit counter. A new table entry has this field initialized to zero before the implied count if it is added by a LOOKUP statement with both ADD and COUNT specified.

.AOMTHIT, .AOMTMISS, .AOMTADD

These attributes are counters. The counters can be accessed using these field names in the &VARTABLE QUERY statement.

.AOMTHIT

Counts the total number of LOOKUP statements, with TOTAL specified, that found a matching table entry.

.AOMTMISS

Counts the total number of LOOKUP statements, with TOTAL specified, that did not find a matching table entry (including those that later added a new entry).

.AOMTADD

Counts the total number of LOOKUP statements, with TOTAL specified, that added a new entry. These are also counted in the .AOMTMIS, since no match is found.

These counters are initialized to zero when a table is allocated or reset.

Chapter 6: Arithmetic in NCL

This section contains the following topics:

About Arithmetic in NCL

Arithmetic NCL statements yield the mathematical result of a calculation and place it in a nominated variable.

Arithmetic is therefore performed as an explicit assignment operation. NCL arithmetic statements are simply a special type of assignment statement.

NCL supports simple or compound arithmetic expressions involving positive or negative numbers. NCL classifies numbers as follows:

- Real numbers that are whole numbers, or numbers containing a decimal fraction. Real numbers can be expressed using scientific notation. The range of numbers supported are those with positive absolute values not higher than +1E+70 and not less than +1E-70, and with negative absolute values not higher than -1E-70 and not less than -1E+70.
- Integer whole numbers only, which are a subset of the real number range, but are not expressed in scientific notation, and lie in the range +2147483647 to -2147483648.

Note: Positive and negative real numbers that lie in the range +1E-70 down to -1E-70 are treated as 0.

Integer Arithmetic

NCL performs integer arithmetic by default. This means that an expression that contains only integers yields only integers as a result.

The problem with integer arithmetic is that it does not allow you to use decimal calculations to get accurate results. The use of integer arithmetic is controlled by the &CONTROL INTEGER operand.

Example: Integer Arithmetic

The following example yields the answer 2:

```
10 / 4 (10 divided by 4)
```

Real Number Arithmetic

NCL treats numbers with decimal points, or expressed in scientific notation, as real numbers and uses floating point arithmetic in the evaluation of real number expressions. The result of a real number calculation, as with integer arithmetic, is placed in a target variable. Unlike integer arithmetic however, the result is not a simple integer but is maintained in the scientific notation form.

The number 22.8 is therefore held in a variable as:

```
+.22800000000000E+02
```

which means:

```
0.228 times 10 to the power of 2
```

A special built-in function, &NUMEDIT (see page 91), is used to reformat real number results into conventional format so that they can be displayed as standard decimal numbers.

&CONTROL REAL

NCL always uses real number arithmetic if the &CONTROL REAL operand is in effect. This means that even if an operation contains only integers, real number arithmetic is used and the result is placed in the target variable in scientific notation form. It also means that calculations are performed accurately, so that whereas, in the example of integer arithmetic given earlier, the result of dividing 10 by 4 is given as 2, with real arithmetic the answer would be the precise answer of 2.5 but expressed in the result variable as:

```
+.250000000000000E+01
```

If you do not code &CONTROL REAL (which means that the default &CONTROL INTEGER is in force), NCL automatically performs real number arithmetic if real numbers are present in any arithmetic expressions. This automatic switch to real number mode is transparent to your arithmetic functions but is of significance if you want to perform comparison operations where real numbers are involved.

Comparisons With Real Numbers

You must use &CONTROL REAL to switch to real number operation if you want to perform comparisons of real numbers. This is because the &IF statement does not know implicitly whether a variable containing a value such as:

```
+.986000000000000E+05
```

is a real number in scientific notation or simply a character string that starts with a plus sign.

Arithmetic Expressions

A simple arithmetic expression in NCL consists of two numbers (real or integer) separated by an arithmetic operator. For example:

```
1 + 2
497021 - 7832
0.08 + 76.889
38 - 97
1E5 - 22.9
44 / 11
```

A number can also be the name of a variable that contains a numeric value, for example:

```
&A + 1
&COUNTER + &INCREMENT
```

The number on the left is operated on by the number on the right. There must be one or more blanks between the operator and the numbers on each side of it.

An expression can also be enclosed in parentheses, in which case blanks are not mandatory between the numbers and the operator. Parentheses can also be used to control the order in which the statement expressions are evaluated.

A compound arithmetic expression consists of two or more numbers and operators, surrounded by parenthesis, which are processed according to the standard rules of precedence to yield a result, for example:

```
(32 + 6) - (7.8 * 2)
```

would yield 38 - 15.6 giving an answer of:

```
+.224000000000000E+02 equivalent to 22.4.
```

Arithmetic Operators

The arithmetic operators are symbols used to specify the operation required when a simple expression is evaluated. The operators used by NCL are, in precedence groups:

******

Exponentiation

/

Divide (REAL number arithmetic)

/

Divide quotient (INTEGER arithmetic)

\

Divide remainder (INTEGER arithmetic)

*****

Multiplication

-

Subtraction

+

Addition

Within each group, the individual operators have equal precedence. The multiplication operator is therefore no more or less significant than the divide operators. Processing of these operators takes place in strict left to right sequence within the expression.

While most of the operators are familiar, the divide options differ depending upon the arithmetic mode that is being used, REAL or INTEGER.

Divide (REAL Arithmetic)

When a division operation is performed in which one or both numbers are real, the result is placed in the target variable as a single real number in scientific notation. The result is not split into quotient and remainder.

Example: Divide

```
&RESULT = (288.5 / 45)
```

yields a value in &RESULT of:

```
+.641111111111110E+01  equivalent to 6.41.
```

Divide Quotient (INTEGER Arithmetic)

When you are using integer arithmetic, the Divide Quotient operator (/) produces division of the leftmost number by the rightmost number and yields the quotient as the result.

Example: Divide Quotient

```
5 / 2 is 2.
```

Note: If you code &CONTROL INTEGER, where either operand is a non-integer real number, the operation is treated as a real number division and yields a real number result in the target variable.

Divide Remainder (INTEGER Arithmetic)

When you are using integer arithmetic the Divide Remainder operator (\) produces division of the leftmost number by the rightmost number and yields the remainder as a result.

Example: Divide Remainder

```
5 \ 2  is 1.
```

This operator is invalid for real number arithmetic and causes a syntax error when encountered.

Divide by Zero

An attempt to divide by zero causes a syntax error and terminates the procedure.

Precedence of Operators

In a simple expression there is only one operator so no precedence considerations arise. The calculation is performed according to the operator.

In a compound expression, the evaluation proceeds from left to right with operators being processed according to the standard rules of precedence. So, in a compound expression such as:

(2 + 3 * 4 ** 2)

there are three operators, which are processed in the order:

******

Exponentiation

*****

Multiplication

+

Addition

The expression is therefore processed in three steps as follows:

```
4 ** 2=16
16 * 3=48
48 + 2=50
```

So an assignment statement coded as:

```
&A = (2 + 3 * 4 ** 2)
```

results in &A being assigned a value of 50.

Parentheses to Control Evaluation Order

Although optional, it is often easier to code an expression in parentheses so that the formula being calculated is easier to read. In fact, you might have to code expressions in parentheses to ensure that your calculation is processed as you intend.

For example, the previous example shows that the result of the compound expression:

```
(2 + 3 * 4 ** 2)
```

is 50.

However, you might have meant something different and coded:

```
((( 2 + 3 ) * 4 ) ** 2)
```

In this case the expression contained in the deepest pair of parentheses is always evaluated first, which causes the calculation to proceed as follows:

```
( 2 + 3 ) =5
( 5 * 4 ) =20
( 20 ** 2)=400
```

which is obviously a completely different result from the one obtained when the same expression was coded without parentheses.

The use of parentheses to delimit the simple expressions within a compound expression is recommended for clarity of understanding.

NCL Substitution and Expressions

Where a variable is to be used as input to an arithmetic expression, the variable should be enclosed in parentheses to ensure that it is evaluated before an operator is applied to it. This is important where the variable contains a signed number.

For example, consider the following:

```
&A ** 2
```

When &A contains 5, the value is 25. However, if &A contained -5, the answer would be -25. This is due to NCL substitution, which evaluates the expression as follows:

```
-5 ** 2 = -(5 ** 2) = -25
```

To ensure that the sign of the variable value is interpreted correctly, parentheses should be used as follows:

```
(&A) ** 2
```

This will give the correct answer of 25.

Signed Numbers

Positive and negative numbers are supported. The result of an expression which yields a negative number carries a minus sign when assigned to the variable that is the target of the assignment statement.

Example: Signed Numbers

```
&A = 5 - 8
```

assigns a value of -3 to &A.

```
&A = (288.5 * 2)
```

assigns a value of +.577000000000000E+03 to &A.

Format Numbers

When you need to display a number, either integer or real, that is held in a variable, you will often have to format the output so that it is aligned correctly in a field or column of numbers.

The &NUMEDIT built-in function lets you reformat any number held in a variable by specifying three parameters:

- The number of characters that the mantissa (the number to the left of the decimal point) is to occupy, padded on the left with blanks if necessary.
- The number of significant decimal positions that is required.
- Optionally, whether the number is to be kept in exponent format.

Examples: &NUMEDIT Usage

In these examples the character ^ represents a blank.

```
&A = (2 * 88) results in ( &A = 176 )
&B = &NUMEDIT (0,2,0) results in &A( &B = 176.00 )
&A = (2 * 88) results in ( &A = 176 )
&B = &NUMEDIT (5,2,0) results in &A( &B = ^^176.00 )
&A = (4.8 + 6.998) results in ( &A = +.117980000000000E+02 )
&B = &NUMEDIT (8,5,) results in &A( &B = ^^^^^^11.79800 )
&A = (2.3 ** 2) results in ( &A = +.529000000000000E+01 )
&B = &NUMEDIT (4,4,E) results in &A( &B = ^^^5.2900E+00 )
```

For more information, see the *Network Control Language Reference Guide*.

Note: The width of the mantissa will always be at least the number of characters in the mantissa, even if the mantissa parameter is less than the actual number of characters. The first example shows this, where the mantissa parameter is 0 but the mantissa length is three characters (176).

The E format of &NUMEDIT always returns one decimal digit to the left of the decimal point.

Chapter 7: Designing Interactive Panels (Panel Services)

This section contains the following topics:

About Panel Services

The Panel Services facility lets you design and implement your own full-screen display formats. Using this facility, you can create NCL procedures for communicating with terminals using full-screen displays, referred to as panels. These panels enable NCL processes to display output data for you, and can be designed with input fields for communicating back to these NCL processes:

- Panel Services can be used by NCL processes executing in any NCL environment associated with a display window.
- Panel Services is supported by an online editor which can be used to create and modify panels.
- Panels can be defined to take advantage of such 3270 features as light-pen and cursor selection. Full-color and extended highlighting support is available on the appropriate terminals.
- Internal editing and validation can be selected for individual input fields to minimize the validation required within NCL procedures.

Logical Screen Manager

Display panels are monitored by a Logical Screen Manager (LSM) which means that only changed data is written to the screen. This ensures that the minimum amount of necessary data is transmitted. Wherever possible, LSM compresses datastreams to ensure they operate efficiently. Using an LSM provides other benefits:

- The peppering effect of data as it is written to some displays is eliminated.
- Screen flashing is minimized.

How You Create or Change Panels

You create and change panels by using an online editor-select option P from the MODS : Primary Menu. You must be authorized to use this facility. Sites can limit the number of concurrent users by restricting the amount of storage available to the online editor.

Note: For more information about using the online editor, see the *Managed Object Development Services Guide*.

When you create a panel, you give it a unique 1- to 12-character name, which is nominated whenever a request is made to display the panel. Once defined, panel details are stored in a VSAM database and can be updated as required.

The standard split-screen facilities are useful when designing panels. The panel editor can be used on one window, while the panel test facility of Edit Services displays the current version of the panel being developed on the other.

Invoke Full-screen Panels

Once defined and saved, a panel can be displayed immediately. Panels for display are usually selected using an NCL procedure containing an &PANEL statement. Other NCL statements such as &LOGON can nominate a panel for display under specific circumstances.

If you try to display a panel that has not been defined, the procedure terminates and displays an error message. You can use the &CONTROL PANELRC statement to receive notification of processing anomalies (including referencing non-existent panels). Your procedure can detect this fact and pre-empt termination to continue processing.

Synchronous and Asynchronous Panel Displays

An NCL procedure can issue an asynchronous panel statement or a synchronous panel statement:

- Asynchronous panel statements provide a panel display, but the procedure continues execution without waiting for input from the terminal.
- If a synchronous panel is displayed, further processing of the NCL procedure stops until you press an Enter key, a Function key, or provide some other input from a light pen or cursor select key. Control then returns to the NCL procedure following the &PANEL statement. The NCL procedure can then process the data just entered from the terminal.

If a synchronous panel has been displayed for a specific length of time without input from an operator, control can be returned to an NCL procedure automatically. The default panel operation is synchronous.

Note: The synchronous state is simpler to understand than asynchronous, and descriptions of panel operations assume synchronous operation unless stated otherwise.

Fixed and Variable Data in Panels

Panels contain a combination of fixed data and variable output data:

- Fixed data is the screen captions, field identification text, and other static screen information defined when the panel is created. This does not change when the panel is displayed.
- Variable output data is data generated by the system when the panel is displayed. It replaces variables positioned within the panel created by the editor. Data is extracted from NCL variables available at the time the panel is invoked.

Variable output data can be displayed in one of the following:

- Protected output-only fields (where the data comprises either system or user variables).
- Unprotected input fields, for any user variables. Once displayed, you can enter data into the unprotected input fields. Panel Services then inserts this data into the user variable for each field, so it is available for further processing by NCL procedures.

Panel Design

Each panel design can contain a series of up to 62 lines, each 80 characters long, and one or more fields. A field character precedes each field. The field character identifies the attributes for that field. These attributes prescribe the following:

- The field type (input, output, selector pen detectable, or null)
- The intensity (brightness) of the display
- Optional editing rules
- The color and extended highlighting used when the field is displayed (for appropriate terminals)

Design Guidelines

When creating or modifying a panel, enter panel details exactly as you want them to display. To avoid confusing the users of your panel:

- Give careful consideration to the use of highlighting and color. Ensure these are not used excessively and do not detract from their effectiveness.
- Use color selection and message placements consistently for all panels in a similar series.
- Clearly identify and caption input fields to describe the input data you require.
- Do not have too many fields on a panel. If necessary, spread data or solicit input over several panels to avoid crowding a single panel.

When a panel is to be displayed, its associated control statements are parsed and any variable substitution performed. This allows control statements to be dynamically tailored.

Field Characters

Each panel line has one or more fields, starting with a field character which prescribes the field attributes.

You must specify a #FLD control statement for each field character your panel requires. There are two ways of defining field characters:

Character mode

To specify character mode, use any special character other than an alpha or numeric character, and excluding ampersand (&), blank, or null.

Hexadecimal mode

To specify hexadecimal mode, enter the hexadecimal value for the character (for example, X'FA'). Use any hexadecimal value in the range X'01' to X'FF', excluding the values X'0E', X'0F' and any values which correspond to the EBCDIC values of characters not allowed on character mode fields. Hexadecimal mode is used where you need a very large number of field types within one panel and there are insufficient special keyboard characters available to accommodate all of the field characters you require.

Note: If using hexadecimal mode field characters, you must prime the panel definition with the preparse option to assign correct values to field character positions in the actual panel.

Field Types

Each field is allocated a field type which prescribes the method for processing the field. Four field types are supported:

OUTPUT

Display only. No data can be entered from the screen.

INPUT

You can both display and enter data.

SPD

Selector pen detectable; data cannot be typed in.

NULL

Display only. Although unprotected, any data entered will be ignored.

Any mixture of these field types can be defined to suit the requirements for a panel you are designing.

The field character that precedes each field determines:

- The field type
- The display characteristics of the field (intensity, color, highlighting)
- (For input fields) the internal validation rules that must be obeyed for data entered in that field. Such rules can specify, for example, that a field is mandatory, must be numeric, cannot contain imbedded blanks, or must be a valid date, and so on.

Each field character you define, occupies the equivalent screen position when the panel is displayed, but appears as a blank character (the attribute byte).

The field proper starts from the next position after the field character, and continues to the next field position on the same line, or to the end of that line where there is no intervening field. Fields do not wrap round from one line to the next.

The three standard default field characters are:

%

high-intensity, protected (no input)

+

low-intensity, protected (no input)

_

high-intensity, unprotected (input, no validation).

These do not require definition by a #FLD statement.

Define any additional field characters you need using the #FLD statement. The attributes for the defaulted characters can be modified. The #OPT statement can be used to nominate alternative standard field characters, so that %, +, and _ can be used within the panel and not processed as field characters.

Column 1 of each line of a panel must be a valid field character; if one is not defined, the attributes for the second standard field character, (normally as +, for low-intensity, protected) are used to replace any data incorrectly placed in that column.

Sample Panels

The following sample panel uses default field characters.

```
%-------------------- Electronic Memo --------------------------
+SELECT  OPTION%===>_SELECT+
%       1+Create a Memo
%       2+Send a memo
%       3+Display incoming memo
%       4+Delete an incoming memo
%       5+Exit from this function
```

In this sample panel, all fields preceded by % display in high-intensity and are protected from data entry. All fields preceded by + display in low-intensity and are also protected. The only field available for input is on the third line, preceded by an underscore (_). The word SELECT identifies the NCL user variable that receives the data you enter in this field once the Enter key is pressed.

By default, the cursor is placed at the SELECT field, as this is the first (and only) field requiring input; no other cursor position has been specified.

Note: The ampersand (&) normally associated with a variable is omitted here.

Assume that our sample panel is called PANEL1. The NCL procedure to display this panel is:

&PANEL PANEL1

When displayed, field characters are removed and the required terminal attribute characters are substituted.

The following sample panel shows how the panel defined in PANEL1 appears, when displayed.

Note: In all figures, the underline symbol (_) designates the cursor location.

```
-------------------- Electronic Memo --------------------------
  SELECT OPTION ===>_
        1 Create a Memo
        2 Send a memo
        3 Display incoming memo
        4 Delete an incoming memo
        5 Exit from this function
```

Override the Input Attribute

Panels generally contain a set of input and output fields which remain relatively fixed. The fields might need to be switched from input to output as a group.

For example, a panel used to add, browse, and update a record has three groups of input fields. They can be:

- Only input during add
- Input during an update (data fields)
- Always input (command fields and so on.)

Separate field attribute characters can be used and switched from TYPE=INPUT to TYPE=OUTVAR.

There are however special cases (field level security for example) where individual fields or unpredictable groups of fields might need to be protected. The &ASSIGN OPT=SETOUT verb can be used to force the protection of a variable, even if it appears in an input field in a panel. A check is made against the current standard field attributes when a field is forced to output. If the attributes of the field match the attributes of one of the standard input fields, the field attributes (such as color and highlighting) are switched to the attributes of the corresponding output field.

Control How Long a Panel is Displayed

Panels do not always have to accept operator input, although this is the most common mode of operation. NCL interfaces to other system components allow you to set up monitoring functions with display-only panels requiring little or no operator entry.

Alternatively, you might want an NCL procedure to resume control if no input is received within a certain time.

By default, whenever a synchronous panel is displayed, Panel Services waits indefinitely for entry from an operator. However, the INWAIT operand of the #OPT control statement lets you override this.

INWAIT lets you specify a time (in seconds or parts of seconds) for Panel Services to wait before control is returned to the invoking NCL procedure. During this interval, standard keyboard entry is accepted and processed normally. Once the time interval expires, control returns to the invoking NCL procedure.

This facility has many applications. For example:

- If a panel displays many high-intensity fields for long periods, you can get screen burnout. Use INWAIT to force a return of control after 20 minutes (for example), when a blank panel replaces the display panel. The original panel can be redisplayed when any input is received.
- If you specify INWAIT=0, keyboard entry is not accepted and control returns to the NCL procedure once the panel is displayed. This could be useful for displaying a message while the NCL processes the last user request.

For more information about the format of the INWAIT operand, see the #OPT statement description in the reference section.

Note: The INWAIT function does not apply to asynchronous panel operations.

Analyze Panel Input

The &INKEY system variable returned to the NCL procedure following a synchronous &PANEL statement, is set to indicate how input was made (for example, by using the Enter key).

After a panel is displayed, Panel Services waits for either of the following conditions:

- For you to complete input (signaled by pressing the Enter key or some other function key)
- For a time-out interval to expire (where an INWAIT period has been specified on the #OPT control statement)

The &INKEY system variable can be tested to find out whether operator input has occurred, and to provide support for Function keys.

If the INWAIT time period elapses, &INKEY is set to a null value. If INWAIT did not expire, &INKEY is set to one of the strings in the following table:

Entry Type	&INKEY Value
Enter key	ENTER
Function key	F01 through F24 (always four characters)
PA key	PA1 through PA3
Light pen	ENTER

Note: If &CONTROL PFKMAP is in effect, F13 to F24 are presented as F01 to F12.

&INKEY can be tested like any other system variable, for example:

```
&IF &INKEY EQ PF01 &THEN &PANEL HELP
```

Remember that where INWAIT is used a null value can be set for &INKEY. Therefore, &IF statements using &INKEY must allow for a possible null value syntax error if &INKEY is eliminated from the statement by variable substitution. This allowance can be achieved as follows:

```
&IF  .&INKEY  EQ  .  &THEN &GOTO .NOINPUT
```

Alternatively, to determine if the INWAIT timer has expired, use return codes from Panel Services as requested by &CONTROL PANELRC. In this case, a return code of 12 is set in the &RETCODE system variable to indicate the INWAIT time interval has expired.

Note: For information about how to simplify testing individual function key values by using direct branching techniques, see the &GOTO verb in the *Network Control Language Reference Guide*.

Example: Panel Input

```
&CONTROL NOLABEL
.DISPLAY
         &PANEL MYMENU
         &GOTO .MENU&INKEY
         .
         .
         drops through if key not supported.
         .
         .
         &SYSMSG = &STR INVALID SELECTION
         &GOTO .DISPLAY
         .MENUENTER-* comes here if ENTER key pressed
         .
         .
         .MENUPF01-* comes here if F1 pressed
         .
         .
         .MENUPF02-* comes here if F2 pressed
         .
         .
         ....
```

The &GOTO .MENU&INKEY statement is resolved as an &GOTO to label constructed by the current value of &INKEY suffixed to .MENU. If the label is not found, the &GOTO acts as a null statement. Control passes to the next statement because the &CONTROL NOLABEL operand was used.

Note: &INKEY is a system variable, so any attempt to assign a value into &INKEY results in an error.

Monitor Panel Return Codes

An NCL procedure can enhance the available synchronous panel processing options by issuing an &CONTROL PANELRC statement before displaying a panel. If this option is used, then:

- After panel processing has completed (that is, following execution of an &PANEL statement) the &RETCODE variable is set to indicate a particular processing condition and control passes to the NCL statement immediately following the &PANEL statement.
- The NCL procedure is responsible for testing the value of &RETCODE immediately after the associated &PANEL statement and processing accordingly.

Note: Failure to test &RETCODE can result in unexpected results.

Asynchronous &PANEL statements always set an &RETCODE completion code, even if &CONTROL PANELRC has not been issued.

The return codes 0, 4, and 8 are set as a result of a process called internal validation, which automatically edits input fields according to rules prescribed in the panel definition. The internal validation process is described fully in subsequent sections.

&RETCODE = 0

For synchronous panel operations: One or more panel input fields have been modified. This return code indicates if any field data have been changed. You could use this to decide if any further validation is required. If data has been overtyped but not altered, it is not considered to have been modified. If internal validation detects an error, return code 8 is set.

If &CONTROL FLDCTL is set and any input field on the panel has MODIFIED attribute, the panel returns &RETCODE 0, even if the user did not physically change the field.

For asynchronous panel operations: The panel has not been updated but input has been received for an earlier display of the same panel. In this case, you can now process the input, but you must reissue the &PANEL statement to update the panel display.

&RETCODE = 4

The same as for return code 0, except that no input fields on the panel have been modified. This return code can be used with return code 0 to decide whether further data processing is required.

Important! Take care when using this return code if multiple interactions with the panel can occur.

If internal validation detects an error, return code 8 is set. Return codes 0 and 4 apply only to the last input from the screen; if the procedure accepts input from a screen where some data is changed and redisplays the panel, which is then not modified, any earlier indications of data modification are lost. Your procedures must allow for this situation.

This return code can be produced for both synchronous and asynchronous panel operations.

&RETCODE = 8

An internal validation error has been detected. When &CONTROL PANELRC is in effect, automatic error condition processing is suppressed so the procedure is notified by this return code. The &SYSMSG variable contains the text of the error message (for example, NOT WITHIN RANGE). The &SYSFLD variable contains the name of the input field in error; this is the name of the variable (minus the & prefix) which receives data entered into that field. When this return code is set, the procedure can ignore the error and take alternative action. For example:

- Displaying a Help panel
- Altering the error message text by assigning a new message to the &SYSMSG variable
- Redisplaying the panel by having the #OPT ERRFLD facility display the error message unchanged and putting the cursor on the field in error. In this case, the ERRFLD operand can be defined on the #OPT statement as ERRFLD=&SYSFLD to simplify processing.

Using &RETCODE = 8 is an ideal way of providing an escape mechanism (such as F3), even though the panel has been defined as having mandatory fields REQUIRED=YES.

&RETCODE = 12

For synchronous panel operations: The panel display time-out limit specified by the #OPT INWAIT operand has elapsed without any data entry. The &INKEY system variable is null.

For asynchronous panel operations: This return code means that the panel has been displayed and has returned to the issuing procedure. Successive updates of the panel can be made by repeating the &PANEL statement; &RETCODE = 12 indicates the panel will be redisplayed and that no input has been received from earlier displays of the same panel. (The availability of input from a previous display of the panel is indicated by return codes 0 or 4.)

&RETCODE = 16

The panel requested is not defined in the current panel library path, or else some other serious error has occurred which prevents the panel displaying. The &SYSMSG system variable contains a message describing the error condition.

If &CONTROL PANELRC is not in effect, the following processing occurs:

- If required, the NCL procedure determines whether any input fields have been modified
- Internal validation handles all error processing automatically and redisplays the panel if errors occur.
- The NCL procedure can only determine whether the INWAIT interval has expired by testing the &INKEY variable for a null value.
- If a requested panel does not exist in the current panel library path, or some other serious error occurs, the NCL procedure terminates with an error message.

Handle Errors

NCL procedures that use Panel Services usually contain processing to validate operator input. If an error is detected, the procedure must be able to notify you about the field in error and the nature of the error.

Errors can be identified by taking advantage of the variable substitution performed for panel control statements, setting appropriate variables for the #OPT control statement CURSOR and ALARM operands, moving suitable text into an error message variable, and redisplaying the panel.

This technique works well, but places additional responsibility on your NCL procedure to ensure that the correct variables are set and cleared later if no errors are detected. (The situation is compounded as the number of fields on the panel increases.)

It also helps the end-user if you highlight those fields in error by switching them to another color, or by using facilities such as blinking or reverse-video on terminals that support these facilities. This can also be achieved by substituting variables in control statements, but becomes unwieldy.

Panel Services facilities are provided to simplify the notification of errors. These facilities assume that error reporting includes additional attention-getting operations such as:

- Ringing the terminal alarm
- Placing the cursor on the field in error
- Displaying error message text
- Changing the field in error to high-intensity
- Switching the field in error to a different color
- Displaying the field in error in reverse-video

The #ERR control statement lets you design a common environment (that is, a common set of attributes) for error conditions. This environment is then applied whenever an input field is nominated as being in error by one of the following methods:

- Using the &ASSIGN verb. (For more information about the syntax and usage of the &ASSIGN verb, see the *Network Control Language Reference Guide*. Note especially the OPT=SETERR and OPT=RESETERR options.) This method lets you set the error attribute for multiple fields
- By nominating the field name in the ERRFLD operand of the panel services #OPT statement. This method allows the error attribute to be set for a single field. The ERRFLD operand normally specifies a variable (for example, ERRFLD=&SYSFLD). When the NCL procedure wants to mark a field as being in error, it places the name of the field in error into the variable (for example, &SYSFLD = FIELD1).

The following sample panel shows the ERRFLD operand in use.

```
#OPT ERRFLD=&SYSFLD
#ERR ALARM=YES INTENS=HIGH COLOR=RED HLIGHT=REVERSE
%--------------------  Name and Address  ----------------------------
%&SYSMSG
+ENTER NAME%===_NAME                    +
+ENTER ADDR%===_ADDR1                   +
+ENTER ADDR%===_ADDR2                   +
+ENTER ADDR%===_ADDR3                   +
+ENTER ADDR%===_ADDR4                   +
+ENTER ADDR%===_ADDR5                   +
```

This sample panel illustrates the use of the #OPT ERRFLD method. This method requires two items in the panel control statements: a #OPT ERRFLD statement and a #ERR statement.

Assume the ADDR5 field data must be XYZ; initially the panel is displayed with the &SYSFLD variable set to null. Because the ERRFLD operand does not nominate a field, the #ERR statement is ignored and the panel is displayed normally.

On subsequent entry, the validation procedure checks the ADDR5 field and determines that it is wrong. The &SYSFLD variable is set to the name of the incorrect field (ADDR5), the appropriate error text is assigned to the &SYSMSG variable, and the panel is redisplayed.

```
.DISPLAY
   &PANEL MYPANEL

   &IF .&ADDR5 NE .XYZ &THEN &GOTO  .ERROR
   .
   .  other processing
   .
.ERROR
   &SYSFLD = ADDR5
   &SYSMSG = &STR DATA MUST BE XYZ, RE-ENTER
   &GOTO .DISPLAY
```

When the panel is redisplayed, the cursor is positioned on the last address field, ADDR5. This field is displayed in high-intensity and the terminal alarm rings (on a color terminal, the field is displayed in reverse-video and red.)

To achieve the same result using &ASSIGN verb instead, remove the #OPT statement from the panel definition and replace the line:

```
&SYSFLD = ADDR5
```

with

```
&ASSIGN OPT=SETERR VARS=ADDR5
```

This method lets you use the #ERR attributes for more than one field.

If the ADDR5 field is corrected and the panel redisplayed, the ADDR5 field returns to its original attributes.

If the erroneous field is a non-display field (such as those used to enter passwords), the overriding attributes that force the entered data to be displayed are ignored and the field remains a non-displayed field.

Note: The variable &SYSFLD is a special variable, which is cleared by Panel Services after the panel is displayed. If you use another variable such as &ERROR, you should clear it after all error conditions are corrected. This ensures that if another panel that uses the same variable is displayed, it does not incorrectly signal an error. The &SYSMSG variable need not be cleared by the NCL procedure because it is always reset to null by Panel Services before returning to the NCL procedure.

Internal Validation

In your own NCL procedure, you can perform all necessary panel field validations. Panel Services also includes powerful automatic editing capabilities for this function, called internal validation.

The Electronic Memo panel definition shown in the previous sample panel does not specify any Panel Services validation for data entered in the SELECT field. You need to define an NCL procedure to validate input and redisplay the panel with an error message.

You can do this by using the following code:

```
&CONTROL NOENDMSG
   .PANEL
      &PANEL PANEL1
      &IF .&SELECT EQ .  &THEN &GOTO .ERR1
      &A = &TYPECHK (NUM) &SELECT
      &IF .&A NE .NUM &THEN &GOTO .ERR2
      &IF &SELECT GT 5 OR +
         &SELECT LT 1 &THEN &GOTO .ERR2
      .
      .  other processing
      .
      .ERR1
         .
      .  build error message and then redisplay the panel.
      .
      .ERR2
         .
      .  build error message and then redisplay the panel.
```

Panel Services can provide internal validation at a field level. This can perform most of the basic editing required for a field and can greatly simplify the processing required within an NCL procedure.

The type of field validation you require can be specified on the #FLD statement for the field character attributes marking the start of a field. Multiple #FLD control statements can be used to specify different validation criteria for individual fields.

The following sample shows a panel that includes the #FLD statement to define validation requirements.

```
#FLD _ REQUIRED=YES BLANKS=TRAIL RANGE=(1,5)
%--------------------  Electronic Memo  ----------------------------
+SELECT OPTION%===>_SELECT + %
&SYSMSG
%    1+Create a memo
%    2+Send a Memo
%    3+Display incoming memos
%    4+Display incoming memo
%    5+Exit from this function
```

In this sample, the standard underscore character (_) has been redefined:

- To specify internal validation for ensuring the field is not omitted (REQUIRED=YES)
- So that trailing blanks but not imbedded blanks can be accepted (BLANKS=TRAIL)
- So that the entered data must be numeric and in a range from 1 to 5 (RANGE=(1,5))

Internal validation error processing depends on how the &CONTROL PANELRC option is set. If &CONTROL PANELRC is not in effect, automatic internal validation occurs; if it is in effect, advanced internal validation can be used.

Automatic Internal Validation

An input error detected by internal validation where &CONTROL PANELRC is not in effect, is automatically handled by Panel Services. Control is not returned to the NCL procedure.

When an error is detected by internal validation, Panel Services assigns the appropriate error text to the &SYSMSG variable, automatically redisplays the panel, rings the terminal alarm, and places the cursor at the field in error. The display options defined within the #ERR statement for the panel are applied to the erroneous field (for example, color or highlighting attributes).

When designing your panel, ensure the &SYSMSG variable field is described somewhere on it:

- Position the &SYSMSG field towards the top of the panel so any message it includes is visible in split screen mode. If &SYSMSG is not provided, the error text cannot display and the operator might not be able to find the cause of error. (The error message texts used by internal validation are detailed in the #FLD control statement EDIT operand.)
- The field for the &SYSMSG variable is normally a high-intensity field, or some prominent color such as red (for color terminals).
- &SYSMSG always resets to a null value when control returns to the NCL procedure if no internal validation is performed and no errors are detected. This ensures an NCL procedure does not have to explicitly clear the &SYSMSG variable if no error is found.

The &SYSMSG variable is not limited to internal validation uses. NCL procedures can use it to display error messages or text assigned during processing.

Internal validation facilities cannot address all of the validation requirements a procedure needs to perform. Like other user variables, the &SYSMSG variable does not span different nesting levels; it is unique to each level unless made available using the &CONTROL SHRVARS option.

Note: Assigning multiple word error text to the &SYSMSG variable requires that you use the NCL &STR or &ASISTR function. For example:

```
&SYSMSG = &STR INVALID SERIAL NUMBER
&SYSMSG = &ASISTR RESOURCE NOT ACTIVE
```

Advanced Internal Validation

While the automatic internal validation greatly simplifies an NCL procedure, sometimes you might want a procedure to continue processing when a validation error is detected, bypassing the automatic reshow normally performed. For example, if a field is specified as mandatory (REQ=YES), automatic internal validation forces the field to be entered and issues this message:

```
REQUIRED FIELD OMITTED
```

Alternatively, you might want a system where a field is mandatory unless F1 is pressed to display a Help panel, or where F12 indicates the entry is to be bypassed. You might also want to change the text of the messages supplied by internal validation.

An &CONTROL PANELRC statement issued before an &PANEL statement tells Panel Services processing that the NCL procedure is designed to cater for a range of &PANEL return codes. These return codes accommodate a variety of conditions possible when processing a panel, and signify there is no automatic reshow performed after an error is detected.

If &CONTROL PANELRC is used, the procedure receives control if internal validation detects that a required field has been omitted (or some other error). The name of the field in error is supplied in a variable called &SYSFLD and the text of the error message registered by internal validation is supplied in the &SYSMSG variable.

In the previous example, where F1 and F12 require special processing, the procedure tests &INKEY for PF01 or PF12, ignores the error, and reacts as required. If F1 or F12 is not pressed, the procedure redisplays the panel with the error message.

When redisplaying the panel, the &SYSFLD variable containing the name of the field in error can be referenced on the #OPT statement ERRFLD operand (ERRFLD=&SYSFLD). This initiates the processing which positions the cursor on the field in error (and possibly a #ERR statement designating special attributes to be applied to the field in error). The text in the &SYSMSG variable can be modified if required, or assigned into another variable.

If a procedure uses this technique, it must be written to handle all return codes possible from the &PANEL statement.

If no internal validation is performed or no internal validation error is detected, &SYSMSG and &SYSFLD are set to a null value when control is returned to the NCL procedure after the &PANEL statement. This ensures that the NCL procedure does not have to explicitly clear variables if an error is not found.

Like &SYSMSG, the &SYSFLD variable is not limited to internal validation uses. It can also be used by the NCL procedure for its own error indications:

- Once a procedure has detected an error it can assign error message text into the &SYSMSG variable, place the name of the field in error in &SYSFLD, and redisplay the panel.
- If the panel uses the ERRFLD option on the #OPT statement, the error message is displayed and the cursor positioned at the field in question. Any other error processing defined on the #ERR statement is also performed.

Like other user variables, the &SYSFLD variable does not span different NCL procedure nesting levels and is unique to each level unless shared under &CONTROL SHRVARS. Regardless of other uses, &SYSFLD resets to null after the &PANEL verb unless an error is detected by internal validation.

Find Out Which Input Fields Have Changed

NCL procedures need a simple mechanism for finding out which input fields have been modified on a display panel with multiple fields. This lets you validate only those fields which were changed.

When an NCL procedure executes with the &CONTROL FLDCTL option set, Panel Services processing automatically creates a stack of all modified input fields returned from the terminal. This stack is built by scanning the input panel line by line from top to bottom, and from left to right. The system variable &ZMODFLD is primed with the name of the input field variable that is logically at the top of the stack. Each time the &ZMODFLD variable is referenced, its value changes to the name of the variable associated with the next modified input field on the panel. When the procedure processes the last panel input field variable name, &ZMODFLD is reset to a null value.

Example: Find Out Changed Input Fields

```
&CONTROL NOLABEL
   .
   .
   .
&PANEL GETABC     -* Display panel containing input fields
                  -* &A, &B, and &C.
   .
   .              -* User enters fields A and C.
   .
   .              -* Procedure resumes processing
   .              -* &ZMODFLD = A
   .
.INPUTLOOP
   &GOTO .&ZMODFLD -* Process next modified field -* variable
   &GOTO .NEXTPANEL -* No more fields...issue next panel.
.A...               -* Process input variable A.  This is
                    -* the first reference to &ZMODFLD,
                    -* whose value now changes to the next
                    -* variable name on the stack, that is, C.
&GOTO .INPUTLOOP
.C...               -* Process input variable C.  This is
                    -* the second reference to &ZMODFLD,
                    -* whose value now becomes null, since C
                    -* is the last modified field variable.
&GOTO .INPUTLOOP
```

An NCL process can have one active &ZMODFLD stack at a time. If another panel is displayed while the &CONTROL FLDCTL option is still in force, then the current &ZMODFLD stack is rebuilt. &ZMODFLD variables that are not accessed remain in the stack if they are in the panel just displayed.

Alternatively, if &CONTROL NOFLDCTL is issued to suspend the &ZMODFLD stack generation, any number of other panels can be presented without destroying the &ZMODFLD stack. The &ZMODFLD stack is available for use unchanged as soon as &CONTROL FLDCTL is reissued. Certain options of the &ASSIGN verb help manipulate the &ZMODFLD stack. For more information about the &ASSIGN verb, see the *Network Control Language Reference Guide*.

Output Padding and Justification

Careful use of padding and justification greatly enhances the usability of panels for end-users. Panel Services includes extensive facilities to manipulate displayed data. Padding and justification qualities are specified by the #FLD statement. There are two justification categories—field level justification and variable level justification. Both can be used concurrently.

Field Level Justification

This is performed on an entire field as delimited by defined field characters. Field justification analyzes the entire field, strips trailing blanks, and pads and justifies the remaining data. The #FLD operands controlling field level justification are JUST and PAD.

The various ways data can be manipulated are best described by a series of examples. These examples show a mix of fields each defined with a different field character and each showing a different display format. Study the #FLD statements and observe the results achieved.

```
#NOTE This sample panel definition gives examples of the
#NOTE use of field level justification and padding.
#FLD#
#FLD$         JUST=RIGHT
#FLD@         JUST=LEFT PAD=<
#FLD?         JUST=RIGHT PAD=>
#FLD/         JUST=CENTER PAD=.
#&VAR01          +
$&VAR02          +
@&VAR03          +
?&VAR04          +
/&VAR05          +
```

Assume the following variable assignment statements are executed by the NCL procedure before displaying the sample panel:

```
&VAR01 = &STR Left justified null padding
&VAR02 = &STR Right justified null padding
&VAR03 = &STR Left justified with padding
&VAR04 = &STR Right justified with padding
&VAR05 = &STR Center justified with padding
```

The default values are JUST=LEFT and PAD=NULL, as shown by the first line in the following example, where field character # is used with no attributes other than the defaults. The sample panel is displayed as follows:

```
Left justified null padding
                              Right justified null padding
Left justified with padding<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>Right justified with padding
..............Center justified with padding..............
```

Variable Level Justification

This operates independently of field level justification, and applies to the data substituted for field variables defined as requiring variable level justification. Variable level justification is designed to help tabulated output, where data of differing lengths is substituted for a series of variables and where the normal substitution process would disrupt display formats. The #FLD operands that control field level justification are VALIGN and PAD.

The substitution process normally substitutes data in place of the &variable without creating additional characters. Thus, if a variable (for example, &VARIABLE) is replaced by data, any characters following this are moved left to occupy any spaces remaining after substitution (that is, if spaces are freed going from a long variable name to a shorter data length).

```
#NOTE This sample panel definition gives examples of the
#NOTE use of variable justification, padding, and field
#NOTE justification.
#FLD # VALIGN=LEFT
#FLD $ VALIGN=RIGHT
#FLD @ VALIGN=CENTER
#FLD ? VALIGN=LEFT PAD=.
#FLD / VALIGN=RIGHT PAD=.
#FLD } VALIGN=CENTER PAD=.
#FLD ! VALIGN=LEFT JUST=RIGHT PAD=.
#&VARIABLE other data+
$&VARIABLE other data+
@&VARIABLE other data+
?&VARIABLE other data+
/&VARIABLE other data+
}&VARIABLE other data+
!&VARIABLE other data+
```

Variable level justification controlled by the VALIGN operand of the #FLD statement, lets you influence the way substitution is performed.

Note: Variable level justification is only performed if the length of the data being substituted is less than the length of the variable name being replaced, including the ampersand (&).

Assume the following variable assignment statement has been executed by the NCL procedure before displaying the sample panel:

```
&VARIABLE = Data
```

&VARIABLE is the only variable within a field which contains the words 'other data'. Where both field justification and variable alignment are used, the padding character applies to both, as shown by the last line of the example for field character !. The sample panel is displayed as follows:

```
Data      other data
     Data other data
  Data    other data
Data.....  other data
.....Data other data
..Data...  other data
...........................Data.....  other data
```

Input Padding and Justification

Fields to which the PAD and JUST operands of the #FLD statement are applied can be defined as input fields. If an input field is primed with data during the display process, the alignment of data within that field when displayed is as described in the previous section on Output Padding and Justification, except that JUST=CENTER is treated as JUST=LEFT. When Panel Services processes input from the screen, input fields defined using the PAD and JUST operands are processed using the following rules:

- Trailing blanks and pad characters are stripped off, unless the pad character is numeric.
- If JUST=RIGHT is specified for the field, then leading blanks and pads are stripped off (including numeric pads).
- If JUST=ASIS is specified for the field, then trailing blanks and pads are stripped off, but leading blanks and pads remain intact.

```
#NOTE      This sample panel definition gives examples of the
#NOTE      use of input padding and justification.
#FLD # TYPE=INPUT
#FLD $ TYPE=INPUT JUST=RIGHT
#FLD @ TYPE=INPUT PAD=< JUST=LEFT
#FLD ? TYPE=INPUT PAD=> JUST=RIGHT
#FLD / TYPE=INPUT PAD=0 JUST=LEFT
#FLD } TYPE=INPUT PAD=1 JUST=RIGHT
#VAR01           +
$VAR02           +
@VAR03           +
?VAR04           +
/VAR05           +
}VAR06           +
```

Assume the following variable assignment statements are executed by the NCL procedure before displaying the sample panel:

```
&VAR01 = Walt
&VAR02 = Tom
&VAR03 = Dick
&VAR04 = Harry
&VAR05 = John
&VAR06 = Vicky
```

The sample panel is displayed as:

```
WALT
                                     TOM
DICK<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>HARRY
JOHN000000000000000000000000000000000000
11111111111111111111111111111111111VICKY
```

If control is passed back to the NCL procedure without any data entered into the input fields, the variables are set to the following values:

```
&VAR01 = WALT
&VAR02 = TOM
&VAR03 = DICK
&VAR04 = HARRY
&VAR05 = JOHN
&VAR06 = VICKY
```

Note: If the line John000... had been modified, it would have been padded to the right with zeros.

The variable values are translated to uppercase because the default for input fields is CAPS=YES.

Process with Light Pens/Cursor Select

Panel Services lets you use both light pens and the cursor select key. Such fields are termed selector pen detectable, or SPD fields.

Specify an SPD field by the TYPE=SPD operand on the #FLD statement. An SPD field can be regarded as an input field and the processing and formatting options apply accordingly. However, you cannot enter data into an SPD field because it is protected.

An SPD field must nominate a single variable (minus the &). This field can contain data to be displayed when the panel is displayed. It is set to the word SELECTED if you do either of the following:

- Press the light pen against the screen anywhere in the field.
- Press the CURSOR SELECT key when the cursor is positioned anywhere in the field.

In accordance with hardware requirements, a field specified as TYPE=SPD must start with either ampersand (&), question mark (?), or a blank ( ). These characters are termed designator characters and have the following meaning:

- A question mark (?) designates a selection field. If you choose this field, the ? is changed to a greater-than symbol (>) by the hardware to show successful selection (it can be deselected by reselecting it once more).
- An ampersand (&) designates the first format for an attention field-select this field in the normal way. (Selecting this field is also equivalent to pressing the Enter key, which transmits all modified screen data.)
- A blank designates the second format of an attention field. It operates like the & field, but modified data is not transmitted.

The designator character can be followed by one or more blanks and then the name of the variable (without the &) that is to receive notification of the selection. For example:

```
#NOTE This sample panel definition gives an example of the
#NOTE use of SPD fields.

#FLD / TYPE=SPD

/? SELECTION1
```

If you use a light pen or the cursor select key to select this field, the variable &SELECTION1 is set to the value SELECTED on return to the NCL procedure.

Mix SPD Fields with Normal Input Fields

While you can mix TYPE=SPD fields with TYPE=INPUT fields, some care must be taken.

A normal input field (TYPE=INPUT) lets you key data into it. This data is transmitted to the system when the Enter key or a Function key is pressed. If an attention SPD field (designated by the blank selection character) is used to replace the Enter key, data entered into normal input fields is not transmitted, and the variables associated with those fields are set to null. Therefore, CA recommends that when you mix SPD fields with normal input fields, you use only SPD fields specifying ? or & designator characters.

Hardware Restrictions

When using a light pen, you must observe some hardware requirements if you define multiple fields on the one line:

- A minimum of three blank or null characters must precede an SPD field.
- A minimum of three blanks or nulls must follow displayed data in the field before the next field starts.

These restrictions do not apply if you are using the CURSOR SELECT key.

Intercept Function Keys

By default, Panel Services intercepts certain function key actions in an identical manner to the rest of the system, as follows:

F2 or F14

Screen split operation

F3 or F15

Terminate NCL procedure

F4 or F16

Terminate NCL procedure

F9 or F21

Screen split/swap operation

While the use of these keys is transparent to NCL procedures invoking full-screen panels, this might not always be desirable. A procedure might want to intercept all function keys and invoke alternative functions.

Panel Services offers two levels of function key interception which can be requested using an &CONTROL verb, before issuing the &PANEL statement where they are to apply. To simplify processing, you can perform optional function key mapping, where F13 to F24 are mapped to their F1 to F12 counterparts.

&CONTROL PFKSTD

PFKSTD stops the interception of F3/15 and F4/16, which normally terminate the invoking NCL procedure. Using these keys returns you to the NCL procedure with the appropriate value set in the &INKEY system variable. F2/F14 and F9/F21 operate as normal, providing screen split/swap facilities.

&CONTROL PFKALL

PFKALL lets you allocate alternative functions to all Function keys, or to block screen splitting, if required. PFKALL stops all Function key interceptions and returns to the NCL procedure with the appropriate value set within &INKEY. In this case, screen split and swap facilities are not available unless your NCL procedure issues the appropriate SPLIT or SWAP command when the associated keys are pressed.

&CONTROL NOPFK

Returns PFKSTD or PFKALL to standard operation.

&CONTROL PFKMAP

This option can simplify the number of keys which a procedure must allow for.

PFKMAP maps F13 to F24 against their counterparts before presenting them to the NCL procedure in the &INKEY system variable. When this option is in effect, the procedure does not have to cater for F13 to F24 because F13 is presented as F1, F14 is presented as F2, and so on.

&CONTROL NOPFKMAP

NOPFKMAP turns off function key mapping.

Function keys are presented to the procedure unchanged, so that all function keys are available for separate uses.

Note: Terminals under EASINET control always operate as though &CONTROL PFKALL is in effect, and cannot operate in split screen mode.

Panels on Different Screen Sizes

The maximum number of display lines that can be defined for a panel is 62. This does not include control statements. It is possible that a panel may exceed the size of the terminal or the size of the current operational window (if operating in split-screen mode). If this happens, Panel Services truncates the panel.

An invoking NCL procedure can use the &LUROWS and &LUCOLS system variables to determine the dimensions of the processing window. The &ZROWS and &ZCOLS system variables can be used to determine the dimensions of the physical terminal. The &ZCURSFLD and &ZCURSPOS system variables are used to determine the actual field which contains the cursor. Using these variables, a procedure can tailor its processing accordingly.

&LUROWS Variable

The &LUROWS system variable is provided to let an NCL procedure test the number of screen lines currently available for displaying a full-screen panel.

When in split screen operation, this is the number of lines available within the current screen window. Use &LUROWS for multi-screen output displays to determine the maximum number of lines that can be displayed in the available operational window.

You are responsible for subtracting any fixed overheads associated with displaying the panel, such as heading lines, and so on, and for operating within the available space. You should also allow for small windows, where there might not be sufficient lines to display data.

&LUCOLS Variable

The &LUCOLS system variable lets an NCL procedure test the number of screen columns currently available for displaying a full-screen panel. &LUCOLS can be used for multiscreen output displays to determine the maximum width that can be displayed in the available operational window.

You should cater for small windows, where there might not be sufficient columns to display data.

&CURSCOL Variable

The &CURSCOL system variable is set on returning from an &PANEL statement and is used to determine the column where the cursor was positioned when the last entry was made.

If operating in split screen mode, the value in &CURSCOL is relative to the start of the operational window regardless of where that window is positioned on your screen. If the last entry was caused by the INWAIT timer expiring, the value returned in &CURSCOL is unreliable.

&CURSROW Variable

The &CURSROW system variable is set on returning from an &PANEL statement and can be used to determine the row where the cursor was positioned when the last entry was made.

If operating in split screen mode, the value in &CURSROW is relative to the start of the operational window regardless of where that window is positioned on your screen. If the last entry was caused by the INWAIT timer expiring, the value returned in &CURSROW is unreliable.

Determine the Field Location of the Cursor

The &ZCURSFLD and &ZCURSPOS system variables are used to determine the actual field location of the cursor. This is useful for providing context sensitive help.

The &ZCURSFLD system variable contains the name of the field the cursor was in. This applies to TYPE=INPUT and TYPE=OUTVAR fields only-output fields have no name.

The &ZCURSPOS system variable provides the offset within that field where the cursor was positioned.

Control Cursor Positioning

The cursor position on a panel is controlled either implicitly by Panel Services (for example, to identify a field in error), or by an explicit request from the procedure issuing the panel.

For explicit cursor positioning, use the #OPT statement CURSOR operand. Either specify the name of an input field defined on the panel, or supply screen coordinates as a row and column number.

- If using the name of an input field, enter the name of the variable minus the ampersand, as defined in the panel to receive any data entered in that field. Define the CURSOR operand as:

  ```
  #OPT CURSOR=&CURSOR
  ```

  Then assign the required field name into the &CURSOR variable before displaying the panel. For example:

  ```
  &CURSOR = FIELD2          -* note omission of ampersand
  &PANEL MYPANEL
  ```

- Alternatively, specify explicit cursor positioning by supplying the row and column where the cursor is to be positioned. The CURSOR operand of the #OPT statement can also be used, but then the row and column coordinates must be specified. For example:

  ```
  #OPT CURSOR=&ROW,&COL
  ```

  Row and column coordinates are then set from the procedure before displaying the panel. For example:

  ```
  &ROW = 10
  &COL = 30
  &PANEL MYPANEL
  ```

Note: If the row and column coordinates that you specify lie outside the boundaries of the current operating window, the cursor is positioned on row 1, column 1 of the window.

A procedure can also influence the implicit positioning of the cursor by Panel Services. Use the &ASSIGN SETERR operand to let the procedure identify one or more fields which can be classified as being in error.

Alternatively, a procedure can use the #OPT panel statement ERRFLD operand to identify a particular field which is in error.

Cursor Positioning Hierarchy

Cursor location is determined in the following sequence:

- The first field in error detected by internal validation
- A field identified as being in error by #OPT ERRFLD
- The first field designated as being in error by &ASSIGN OPT=SETERR

  Any field designated by &ASSIGN OPT=SETERR that is also identified by #OPT CURSOR takes precedence.
- A field or position identified by #OPT CURSOR=*loc*
- The first input field processed top to bottom, left to right
- The upper left corner of the panel (row 1, column 1 of the window)
- If the selected field cannot be displayed within the current window dimensions, the upper left corner of the panel (row 1, column 1 of the window)

Dynamically Alter Panel Designs (PREPARSE)

When a panel is designed, the location and layout of input and output fields within it is normally fixed when the panel is created using Edit Services. This means the field characters that define the location of fields (by default %, +, or _), are positioned as required and remain fixed. It is the data or variables within those fields that then change.

However, under some circumstances, you might need to dynamically alter the attributes of fields, or to add or delete entire fields. For example, you might want the color and highlighting for a field to change depending on the data content. (This might be necessary if you are designing a panel that monitors network status, where you want to vary the color of a field to alert an operator).

Some scope for doing this exists by supplying variable data for use on #FLD statements where the variable data is used to alter the characteristics of the field. However, the actual addition or deletion of fields cannot be achieved using this technique and it becomes cumbersome if the attributes of many fields must be altered.

Panel Services lets you dynamically create panel definitions by using a preparsing concept. Preparsing is requested by the #OPT panel statement PREPARSE operand, and makes a preliminary scan of the required panel lines before building the panel.

During this preliminary phase, field characters are ignored and substitution used to change a panel line in any way you require. Only after preparsing is complete, is the normal panel building process performed.

Substitution normally uses the occurrence of ampersand (&) to indicate the start of a variable string. These strings can be resolved to reflect the content of the variable, as set by the procedure before the &PANEL statement.

The PREPARSE operand has the following format:

```
#OPT PREPARSE=($,S)
```

```
#OPT PREPARSE=(!,D)
```

The first character in the parentheses defines the alternative substitution character (other than an &) to be used during the PREPARSE operation.

The second character specifies the alignment option for the display fields affected by PREPARSE substitution. These options (for Dynamic and Static preparsing) are explained in the following sections.

The alternative substitution character (for example, a dollar sign) is used as the substitution character for the preparse stage of processing. The panel line is scanned for occurrences of the preparse character and the variables are isolated. The variables are then resolved using the values of the corresponding variables set from the procedure. Using an alternative substitution character allows panels to contain a mix of conventional and preparse fields.

Before displaying this panel (see the following sample, which shows the panel before PREPARSE substitution), the procedure tests the values in the variables &FIELD2 and &FIELD3 and appends the appropriate field character to display the field. In this example, using the > character displays the field in red and reverse video, but using the ? character displays the field in yellow without any other highlighting.

```
#OPT PREPARSE=($,d) INWAIT=60
#FLD > COLOR=RED HLIGHT=REVERSE TYPE=OUTPUT
#FLD ? COLOR=YELLOW TYPE=OUTPUT
%--------------------  Network Monitor  -------------------------
+$FIELD1

  +NETWORK STATUS AS AT &TIME ON &DATE3
  +NCP1 is currently ...........$FIELD2

  +NCP2 is currently ...........$FIELD3
```

In this example, FIELD2 and FIELD3 are output fields. FIELD1 is an output field too, by default, but the logic below also converts the FIELD1 position to an input command line if the user ID happens to be USER01. The value assigned to FIELD1 actually contains the new field characters that are to be substituted into the panel definition. The preparse function then builds the new fields before the panel is displayed, so that an input field appears on the terminal.

The following logic shows this procedure:

```
&IF &STATUS2 EQ INACT &THEN &FIELD2 = >&STATUS2
&ELSE &FIELD2 = ?&STATUS2
&IF &STATUS3 EQ INACT &THEN &FIELD3 = >&STATUS3
&ELSE &FIELD3 = ?&STATUS3
&IF &USERID = USER01 &THEN &FIELD1 = &STR Command +
     %===>_COMMAND
&PANEL MYPANEL
```

Note: Although the variables on the panel start with the preparse substitution character ($), they are still referred to in the NCL procedure as starting with an ampersand.

Preparsing is regarded as a completely separate substitution phase. Therefore, if a preparse character other than an & is designated, this process can substitute data that includes other variables, which are resolved when the standard substitution process for each field is performed. The following sample show a panel after PREPARSE substitution.

```
#OPT PREPARSE=($,D) INWAIT=60
#FLD > COLOR=RED HLIGHT=REVERSE
#FLD ? COLOR=YELLOW
%--------------------  Network Monitor  -------------------------
+Command %===>_COMMAND

+NETWORK STATUS AS AT &TIME ON &DATE3
+NCP1 is currently ...........>INACT

+NCP2 is currently ...........?ACT
```

Dynamic PREPARSE Option

In panel lines with more than one preparse character or field character, the effect of substituting variable data within the line might cause the final text of the line to be longer or shorter than the original text. In this case, fields that follow the substituted data move to the left or right of their original column.

Shifting preparse or field characters to accommodate differing substituted data lengths is the system default, and is called dynamic preparsing.

Static PREPARSE Option

If you want to display a panel where column alignment is important for the presentation, but you are using preparse to substitute data into each line, you could find the columns are not aligned properly in the final display. This occurs because the substituted data varies in length from line to line.

To correct this, use the static preparse option. This lets you specify on the #OPT statement that the location of preparse and field characters is to be preserved, despite differing data substitution lengths. Here, data is truncated to fit if the substituted data exceeds the space available to the left of the next preparse or field character.

Example: PREPARSE Option

If a panel uses $ as its preparse character and contains the following line:

```
$VAR1  $VAR2
```

An NCL procedure sets the variable as follows:

```
&VAR1 = &STR !ABCDEFGHIJKLMNOPQRSTUVWXYZ
&VAR2 = &STR !12345
```

Dynamic preparse displays:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ 12345
```

Static preparse displays:

```
ABCDE 12345
```

Considerations When Using PREPARSE

Preparsing substitutes by using the specified character. After the preparse is complete, standard panel field processing proceeds.

Take care that the data substituted does not contain unwanted field characters which introduce unexpected fields if not removed. Preparsing may generate an entire line of data that is expected to be output only.

If this data contains any underscore characters (indicating an input field) errors will probably occur because the field format is incorrect. Overcome this by ensuring that the data substituted by preparsing does not contain such unwanted field characters. Varying the default field characters can help here.

Allowing the panel to perform substitution after preparse prevents this problem. For example, change the NCL code as follows:

```
&IF &STATUS2 EQ INACT &THEN +
    &FIELD2 = &CONCAT > &STATUS2
```

Display Function Key Prompts

The SAA Common User Access (CUA) standards require that a list of function keys and their functions be displayed at the bottom of a panel. The #TRAILER control statement can be used by NCL procedures to nominate lines which appear at the bottom of the panel. The function key prompts are then always displayed at the bottom of the panel regardless of the screen size.

Control the Formatting of Input Fields

When a panel is displayed, a data stream is created to format the physical screen as defined in the panel definition. For example, a panel may be displayed repeatedly as a monitor screen or similar, where the display time is controlled by the INWAIT operand. In this example, the INWAIT timer can expire while data or a command is still being entered in the panel. The entered data is ignored and the input field cleared when the panel is refreshed. The cursor position might also change.

You can use the #OPT FMTINPUT statement to avoid this problem by letting the procedure determine when input fields are formatted. So, if you are entering data when the screen is refreshed, and the FMTINPUT=NO specification is used, the entered data remains and you can complete entry unaffected.

By varying the FMTINPUT operand setting (through a variable set from within the procedure) from YES to NO, the procedure can toggle input field formatting on and off. For example:

```
#OPT FMTINPUT=&FMT
```

The first time a panel is displayed, it should always specify FMTINPUT=YES to ensure that the physical screen is correctly formatted. On subsequent refreshes (usually after INWAIT expires), FMTINPUT=NO can be used.

For example:

```
.FMTYES
     &FMT = YES
     &GOTO .DISPLAY
.FMTNO
     &FMT = NO
.DISPLAY
     &PANEL MYPANEL
     &IF .&INKEY EQ .  &GOTO .FMTNO
     .
     .
     .  standard processing
     .
     .
     &GOTO .FMTYES
```

Note: The FMTINPUT operand is designed to work in conjunction with the INWAIT facility and must be used with care, or panel errors can result.

Allow Long Field Names in Short Fields

Panel Services panel definition screens look similar to the panel which is to be displayed. The panel definition screen contains attribute characters followed by the field data (for TYPE=OUTPUT fields) or variable names (for TYPE=INPUT or TYPE=OUTVAR fields). The next field attribute character is placed at the start position of the next field. Unfortunately this means that input fields cannot be any shorter than the variable name which is to contain the input data.

To overcome this, you can use the #ALIAS control statement to define a short alias name for a variable. The alias name can be used where the variable would have been used. You can define a range or list of variables and refer to them by the same alias name in the panel definition.

Retrieve Panels from Panel Libraries

Panels are created using an online editor and are stored in a panel library. When a user is defined, the panel library, or sequence of panel libraries that they are to use, is defined. This is called the panel library path. When required, panel specifications are retrieved from a library in the user's current path.

To eliminate overheads associated with retrieving the panel from the library, an in-storage queue of active panels is maintained. When a panel is first referenced, it is retrieved from a panel library and stored on the active panel queue.

Thereafter, the panel is retrieved from the active panel queue without reference to the panels library. If one of these panels is modified (using the online editor), the old copy is removed from the active panel queue so that the next reference retrieves the updated panel from the panel library path.

Note: If a panels library is being shared by more than one system, the old version of a modified panel is only removed from the active panel queue of the system on which the panel change has been made. The other systems continue to use the old panel until it is rolled off the active panel queue by other panels being used in the system. The LIBRARY REFRESH command can be used to drop all panels loaded from a library from the active panel queue.

The SHOW PANELS command can be used to list the panels that are retained in the active panel queue.

Display Panels on OCS Windows

Full-screen displays can be invoked from an OCS window. OCS windows normally operate in roll mode from top to bottom of the window.

Any NCL process executing in the NCL processing environment associated with an OCS window can issue &PANEL in an attempt to take over the window and place it in full-screen mode.

OCS always grants this bid for the display area if it is operating in its usual rolling mode. It places the window in full-screen mode and initiates queuing of any messages that are directed to the OCS window while it is operating in full-screen mode.

On completion of the NCL procedure that has taken over the window, or when the procedure issues an &PANELEND statement, the OCS window reverts to standard roll mode operation and any queued messages are displayed.

If the number of queued messages exceeds the size of the OCS window, the window is automatically placed in AUTOHOLD mode to ensure you have time to observe all messages.

NCL Processes Competing Against OCS for the OCS Window

If an NCL process executing in an OCS window's NCL processing environment issues an &PANEL statement while the OCS window is in its usual roll mode, OCS (the current owner of the window) always allows the panel to be displayed.

If an NCL process attempts to issue an &PANEL statement when the OCS window is in Holding or Autohold On mode, the &PANEL statement is suspended. The OCS window is placed into FS-HOLD mode and requires operator input before the NCL process panel is allowed to take over the window. At this stage, (Holding or Autohold) the OCS window is logically considered to be in an &CONTROL NOSHAREW condition. That is, OCS owns the presentation area and is not prepared to release it.

Operator input releases the Holding or Autohold mode; OCS immediately switches to the logical &CONTROL SHAREW condition and allows the panel to be displayed.

Competition Between NCL Processes for an NCL Environment Window

Many NCL processes can execute concurrently in an NCL processing environment. If the NCL environment is associated with a window, any process can issue &PANEL in an attempt to take over the window's presentation area to show you their particular panels.

At all times there is a logical owner for the presentation area represented by the window. For an OCS window this is OCS itself most of the time, but when an NCL process issues &PANEL it acquires logical ownership of the window and OCS operates in the background.

The same applies for other NCL processing environments. For example, the Primary Menu is initially owned by the Primary menu NCL process. If that process STARTs other NCL processes which then execute within the same NCL processing environment, these other NCL processes can bid for the window to display their own panels by issuing an &PANEL.

Once a process succeeds in becoming the (temporary) owner of a window, it can indicate its willingness to give up its ownership through the &CONTROL SHAREW option.

If the current owner of the window is executing with &CONTROL SHAREW in effect, it will automatically give up its ownership to any other process that issues &PANEL. The previous owner is then logically stacked behind the new owner. When the new owner ends or issues &PANELEND the previous owner's panel is redisplayed.

One of the most common usages of asynchronous NCL processes competing for a window is in an OCS environment. A user can have several independent NCL processes executing in the NCL processing environment associated with an OCS window. These processes might monitor the status of various items in the network and report to the operator when an error condition occurs, by issuing an &PANEL statement to takeover the window.

These processes are written to use &CONTROL NOSHAREW, preventing any other process from stealing the window from them once they displayed their panel. On acknowledgement of their panel by the operator the process issues &PANELEND to release their ownership. The window then reverts to the next waiting process or to OCS.

Note: If a panel is defined with INWAIT=0, indicating that the issuing procedure regains control immediately the panel is displayed, the system guarantees to display the panel before the issuing procedure is allowed to continue. This might mean waiting for some other process to release control of the window.

Asynchronous Panels

NCL lets you write procedures that interact with a display area either synchronously or asynchronously. If you operate in synchronous mode, when a screen panel is issued the procedure is suspended until input is received from the device or until a timeout (as specified by INWAIT) expires.

This mode of operation suits applications where direct interaction with an operator is expected. The INWAIT function also allows interval-based dynamic updating of screen displays.

Synchronous mode operation does not suit applications which require dynamic updating of screen displays as events occur, or circumstances in which screen displays might have to be updated without necessarily involving any operator input.

To solve these requirements there is a mode of panel operation called asynchronous operation.

Asynchronous Operation Concepts

Asynchronous panel operation lets you write an NCL procedure that can issue a panel for display but then continue execution without being suspended to wait for operator input or timeout expiry.

This means that you can develop procedures to drive dynamically updated event panels to display new event information as it happens (rather than at fixed intervals) while retaining the ability to interact with an operator. It also lets you update operator information screens at any time without requiring operator input to trigger resumption of processing.

Invoke an Asynchronous &PANEL Operation

To issue a panel in asynchronous mode, code the TYPE=ASYNC operand on the &PANEL statement. Your procedure is suspended until the nominated panel is scheduled for display, and then control returns from the procedure to the statement immediately following the &PANEL statement.

At this stage, the system variable &RETCODE is set to indicate the result of the &PANEL statement.

The values in &RETCODE are equivalent to the return codes set if you specified &CONTROL PANELRC.

&RETCODE can contain one of the following values:

0

The display operation has not been performed because the nominated panel has already been displayed and input is now available from that earlier display.

This happens if your procedure repeatedly updates a panel but, between updates, the operator enters some input into a field on the panel and causes an input operation by pressing Enter or a function key. The input variables defined for the panel now contain the updated values as entered by the operator.

4

As for &RETCODE = 0, except no input fields have been changed, so the operator has only pressed Enter or a function key. Again, the display operation does not take place.

12

The display operation has been scheduled. The panel is displayed either for the first time or to update an earlier display of the same panel.

The meaning of &RETCODE = 12 is completely different from the same return code if a synchronous &PANEL statement is issued.

Waiting for Input

Long-running NCL procedures do not execute continuously. At some point in their logic they go into a wait state, waiting for some event to occur that triggers them to start processing again. A typical example of such an event is input received from the operator.

With synchronous &PANEL operation, the &PANEL statement implies a wait-for-input condition. With asynchronous &PANEL operation, the panel statement itself does not wait for input from the terminal, so another mechanism is needed which tells the procedure when input has been received. The mechanism used is &INTREAD and the procedure's dependent request queue (see page 42).

When an asynchronous panel is displayed and something happens on the window (for example, the operator enters input to the panel), a special message is delivered to the procedure's dependent request queue in the format:

```
N00101 NOTIFY: PANEL EVENT: event-type RESOURCE:panelname
```

where event-type describes the type of input activity that is being notified.

When the procedure reaches a point in its logic where it needs to wait for input from the terminal, it issues an &INTREAD TYPE=REQ statement. This automatically places it in a wait state until a request message arrives on its dependent request queue.

When input is entered by the operator, the N00101 message is delivered to the dependent request queue, satisfies the &INTREAD statement and the procedure wakes up. By examining the contents of the N00101 message the procedure can determine that it is a notification of a panel event occurring on the asynchronous panel.

It is important to note that the message is only a notification that a panel event has occurred. The action taken by the NCL procedure after reading the message depends on the event. The possible events and associated actions are:

INPUT

Input has been received from the terminal. For the input to be made available to the NCL procedure in the associated panel input variables, it is necessary for the procedure to issue an &PANEL TYPE=ASYNC statement.

CHANGE

The terminal window conditions have been changed. This could indicate that a redefinition of the window dimensions has occurred (a SPLIT or SWAP operation has taken place).

BCAST

A broadcast has been sent to the panel. For the broadcast to appear on the asynchronous panel currently displayed, it is necessary for the procedure to issue another &PANEL statement.

TAKEOVER

Another NCL process has attempted to take over the window. The NCL process is only notified of this event if it is the window owner and &CONTROL NOSHAREW is in effect. To allow another NCL process to acquire ownership of the window, the NCL procedure can issue a &PANELEND or a &CONTROL SHAREW statement.

If an INPUT, CHANGE, or BCAST event occurs, the NCL procedure very often issues an &PANEL statement to obtain the input to the asynchronous panel. This statement must specify the name of the panel being displayed when the &INTREAD statement was issued. The &PANEL statement completes with the appropriate &RETCODE return code value, as described previously. If an &PANEL statement specifying another panel name is issued after the N00101 message is read, the input is no longer available.

Coordinate Other Processing with Input Notification

The fact that input from a panel can be notified not by completion of the &PANEL statement but instead by the arrival of a trigger message on the procedure's internal request queue means that a procedure has the ability to wait for more than one source of input to act as a trigger for it to resume processing.

For example, consider a procedure that has the role of maintaining a dynamically updated screen display which has to be refreshed with new information as soon as it arrives, but which can receive an input command from the screen.

Such a procedure needs to be able to listen for new information that is to be displayed as well as listening for input received from the screen display.

By designing the procedure so that it receives all its input via &INTREAD TYPE=ANY, it can receive new display messages from its dependent response queue (for example from other procedures operating in its dependent environment) and notification of screen input via its dependent request queue.

This concept therefore allows a procedure to have a single point in its logic at which it can wait for multiple different forms of input, and so synchronize its processing of different input streams.

Control Input Field Initialization

When displaying an asynchronous panel, it is possible to receive output directed to the panel while input is being entered. This could result in partially updated panel input fields being reset when the panel is redisplayed to reflect the received output.

The FMTINPUT operand of the #OPT statement in a panel definition determines if input fields are formatted when a panel is displayed. This operand can be used to control input field re-initialization for both synchronous and asynchronous panels.

In the case of asynchronous panels, if the FMTINPUT operand is not specified in the panel definition, the input fields are reset only the first time the panel is displayed or when the panel is redisplayed after the operator has signaled the end of input (by pressing Enter or a defined Function key). This means that input can be entered continuously by the operator while the panel output fields are being rewritten as output is directed to the terminal. If the FMTINPUT operand is specified in the panel definition, input field initialization is processed as defined by the operands to #OPT statement.

Manage I/O Contention

Using the CDELAY operand of the &PANEL statement, it is possible to prevent output to a terminal in input mode being delayed, irrespective of the value of the SYSPARMS CDELAY operand. This is advisable in situations where output can affect the structure of a panel being displayed.

For example, members might be asynchronously inserted into a selection list while selections are being made by the operator. In this situation, the chosen selection variable may be associated with a certain entry before the list is updated and a associated with a different entry afterward. Specifying CDELAY=N synchronizes the arrival of the output with its appearance on the panel.

For more information about the CDELAY concept, see the *Reference Guide*.

Panel Control Statements

Optional control statements can precede a panel to specify the particular requirements for that panel. These are:

#ALIAS

Defines alternative field names

#ERR

Defines the action to be taken for an error condition

#FLD

Defines or modifies a field character's attributes

#NOTE

Provides installation documentation (this is ignored during processing)

#OPT

Defines optional operational requirements

#TRAILER

Defines trailing lines for the panel

Control statements included within panel definitions must precede the displayable portion of the panel which is determined by the first line that is not a control statement. Control statements must start in column 1 of the lines on which they appear.

Note: You cannot include comments on the same line as a control statement (except #NOTE).

#ALIAS Control Statement—Define Alternative Name for Input Variables

This control statement allows the panel definition to contain alternative names for variable names in TYPE=INPUT and TYPE=OUTVAR fields.

This facility is useful if you want to have short fields with long variable names. Each reference to name in the panel definition is regarded as a reference to the next name from the list of VARS specified.

This control statement has the following format:

```
#ALIAS name
    {  VARS=prefix*[ RANGE=(start,end) ] |
       VARS={ vname | (vname,vname, …,vname) } }
```

name

Specifies the alias name that appears in the panel definition. Whenever this name occurs in a field declared as TYPE=INPUT or TYPE=OUTVAR on the #FLD statement, Panel Services logically replaces it with the next available name from the VARS list.

The name can be from one to eight characters in length. The first character must be an alphabetic or national character. The remaining characters, if any, must be alphanumeric or national characters.

The same name can be used on multiple #ALIAS statements. The variable names are simply added to the end of the list of names to which the alias name refers.

VARS=*prefix [RANGE=(*start*,*end*)] |**

VARS=(*vname*,*vname*, ..., *vname*)

Specifies the list of names to replace the alias name in the panel definition. Each time the alias name is encountered in the panel definition, it is replaced by the next available name from this list. The formats of the operands associated with VARS= are as follows:

- VARS=*prefix** denotes that the variable names used are *prefix*1, *prefix*2, and so on. The RANGE= operand can be specified to indicate a starting and ending suffix number. The default is RANGE=(1,64). The format *prefix** cannot be used with other variable names on the same #ALIAS statement.
- VARS=*vname* is the name of a variable, excluding the ampersand (&).

Examples: #ALIAS Control Statement

```
#ALIAS Z VARS=LONGNAME
#ALIAS Z123 VARS=(SATURDAY,SUNDAY)
#ALIAS AVAR VARS=LINE* RANGE=(10,20)
```

Notes:

- Multiple #ALIAS statements can be used for the same alias name if insufficient space is available on a single statement.
- If an alias name appears in the panel definition after all the variable names in the alias list have been used up, the alias name itself appears in the panel.
- Symbolic variables can be included in the #ALIAS statement. Variable substitution is performed before the statement is processed, using variables available to the NCL procedure at the time the &PANEL statement is issued.

#ERR Control Statement—Define Action Taken During Error Processing

This control statement determines the processing required when a panel is being redisplayed following an error condition.

An error condition can be detected either by Panel Services internal validation or by the processing NCL procedure. If detected by internal validation (and &CONTROL PANELRC is not in effect), Panel Services invokes error processing automatically. If detected by the processing NCL procedure, error processing is invoked in one of two ways:

- By using the &ASSIGN OPT=SETERR verb
- By nominating the name of the variable that identifies the invalid input field on the ERRFLD operand of the #OPT statement

  This is dynamically invoked by using a symbolic variable with the ERRFLD operand and setting this variable to the name of the variable (minus the ampersand) that identifies the field in error.

When #ERR processing is initiated, the cursor is positioned to the first field in error. The panel is redisplayed, applying the attributes defined on the #ERR statement to the fields in error. This technique provides the panel user with a simple means of drawing the terminal operator's attention to the field in error. This is particularly effective on color terminals where the color of any field in error can be altered and reverts to normal when the error condition is rectified.

Normally only one #ERR statement is defined. However, if required to accommodate the operands, multiple statements can be defined. They can be defined in any order. However, as with #OPT, #FLD, and #NOTE statements, any #ERR statement must precede the start of the panel, as determined by the first line that is not a control statement.

This control statement has the following format:

```
#ERR [ INTENS={ HIGH | LOW } ]
   [ { COLOR | COLOUR }={ BLUE | RED | PINK | GREEN |
                    TURQUOISE | YELLOW | WHITE | DEFAULT } ]
   [ { HLIGHT | HLITE }={ USCORE | REVERSE | BLINK | NONE } ]
    [  ALARM={ YES | NO } ]
```

INTENS={ HIGH | LOW }

Determines the intensity of the error field when displayed. The INTENS operand is ignored for terminals with extended color and highlighting when either the COLOR or HLIGHT operand is specified.

HIGH

Specifies that the field is displayed in double intensity.

LOW

Specifies that the field is displayed in low or standard intensity.

**{ COLOR | COLOUR }={BLUE | RED | PINK | GREEN |
TURQUOISE | YELLOW | WHITE| DEFAULT}**

Determines the color of the field. The operand applies only to IBM terminals with seven-color support and Fujitsu terminals with three- or seven-color support.

If the terminal does not support extended color, the COLOR operand is ignored. This feature enables COLOR to be specified on panels that are displayed on both color and non-color terminals. COLOR can be used with the HLIGHT operand.

For Fujitsu terminals that support extended color data streams, but support only three colors, the following color relationships are used:

Specified	Result (on Fujitsu three-color terminal)
GREEN	GREEN
RED	RED
PINK	RED
BLUE	GREEN
TURQUOISE	GREEN
YELLOW	WHITE
WHITE	WHITE
DEFAULT	GREEN

Fujitsu seven-color terminals are treated the same as IBM seven-color terminals.

The DEFAULT keyword indicates that the color of the field is to be determined from the INTENS operand. This feature is useful if you want to set the color from an NCL procedure (that is, COLOR=&COLOR is specified and the NCL procedure can set the &COLOR variable to DEFAULT).

{ HLIGHT | HLITE }={ USCORE | REVERSE | BLINK | NONE }

Applies only to terminals with extended highlighting support, and determines the highlighting to be used for the field.

The HLIGHT operand is ignored if the terminal does not support extended highlighting, so HLIGHT can be specified on panels that are displayed on terminals that do not support extended highlighting. HLIGHT can be used with the COLOR operand.

When NONE is specified, the HLIGHT operand is ignored and no extended highlighting is performed for this field.

ALARM={ YES | NO }

Determines whether to ring the terminal alarm when the panel is displayed with an error condition. This alarm works independently of the ALARM operand on the #OPT control statement.

Examples: #ERR Control Statement

```
#ERR COLOR=RED HLIGHT=REVERSE ALARM=YES
#ERR COLOR=YELLOW HLIGHT=BLINK INTENS=HIGH
```

Notes:

- Only those attributes defined on the #ERR statement are modified for the field in error. All other attributes associated with the original field, such as internal validation, remain intact.
- Symbolic variables can be included in a #ERR statement. Variable substitution is performed before processing the statement, by using variables available to the NCL procedure at the time the &PANEL statement is issued.
- When &CONTROL PANELRC is in effect, internal validation does not automatically reshow the panel with the error message, and so on. In this case, the procedure regains control following the &PANEL statement with the &RETCODE system variable set to 8 to indicate that an error has occurred. The &SYSMSG variable contains the text of the error message that describes the error and the &SYSFLD variable contains the name of the field in error. This name is the name of the variable in an input field that would receive the data entered into that field.
- The &ASSIGN statement SETERR option provides a mechanism for assigning #ERR field attributes to multiple (input field) variables before displaying a panel. This feature lets you accept input from a number of different fields on a panel, validate all the fields, and redisplay the panel with all incorrect fields displayed with the #ERR attributes. The user sees all the errors at one time, rather than field by field.

#FLD Control Statement—Define or Modify a Panel Definition Field Character

This control statement tailors the operational characteristics of a panel.

When a panel is defined, it is made up of a number of lines, which in turn are made up of a number of fields. Each field commences with a field character, which appears as a blank on the panel when displayed. Each field character determines the attributes to associate with the field following the field character itself. A field is delimited by the next field character or the end of the panel line. Fields cannot wrap from one line to the next.

The first field on a line always starts in column 1. If no field character is defined in the first position of the line, the attributes of the second of the three standard field characters are forced. These attributes are usually a plus sign (+), TYPE=OUTPUT, and INTENS=LOW. They replace any non-field character incorrectly placed in this position.

Before parsing, the #FLD statement is scanned and variable substitution is performed. This process makes it possible to tailor dynamically any of the options or operands on the statement.

You can specify as many #FLD statements as required, and you can define them in any order. However, as with #OPT, #ERR and #NOTE statements, all #FLD statements must precede the start of the panel, as determined by the first line that is not a control statement.

This control statement has the following format:

```
#FLD { c | X'xx' }
     [ BLANKS={ TRAIL | NONE | ANY } ]
     [ CAPS={ YES | NO } ]
   [ { COLOR | COLOUR }={ BLUE | RED | PINK | GREEN |
             TURQUOISE | YELLOW | WHITE | DEFAULT } ]
     [ CSET={ ALT | DEFAULT } ]
     [ EDIT={ ALPHA | ALPHANUM | DATEn | DSN | HEX |
            NAME | NAME* | NUM | REAL | SIGNNUM | TIMEn } ]
   [ { HLIGHT | HLITE }={ USCORE | REVERSE | BLINK | NONE } ]
     [ INTENS={ HIGH | LOW | NON } ]
     [ JUST={ LEFT | RIGHT | ASIS | CENTER | CENTRE } ]
     [ MODE={ SBCS | MIXED } ]
     [ NCLKEYWD={ YES | NO } ]
     [ OUTLINE={ {L R T B} | BOX } ]
     [ PAD={ NULL | BLANK | char } ]
     [ PSKIP={ NO | PMENU } ]
     [ RANGE=(min,max) ]
     [ REQUIRED={ YES | NO } ]
     [ SKIP={ YES | NO } ]
     [ SUB={ YES | NO } ]
     [ TYPE={ OUTPUT | INPUT | OUTVAR | SPD | NULL } ]
     [ VALIGN={ NO | LEFT | RIGHT | CENTER | CENTRE } ]
```

***c* | X'*xx*'**

Specifies the field character that is used in the panel definition to identify the start of the field.

c

Specifies a *single* character that is *not* alphanumeric. Any special character (for example, an exclamation mark) can be used, except an ampersand (&), which is reserved for use with variables.

X'*xx*'

Specifies the hexadecimal value of the field character. Use this notation to specify any value in the range X'01' to X'3F'. Do not use values which correspond to alphanumeric characters, or X'0E' (shift in) or X'0F'(shift out).

Although the panel editor prevents you from entering non-displayable hexadecimal attributes (X'01' to X'3F') in the body of the panel, you can use the PREPARSE option to prime the field character value in the panel before issuing the &PANEL statement.

The first #FLD statement to reference a particular field character defines a new character. Subsequent statements referencing that same field character modify or extend the attributes of the field character. Three standard field characters (%, +, _, unless altered by the #OPT statement) are provided. If a default field character (usually % + or _) is referenced, it is the same as extending or modifying the attributes of an existing field character.

If no additional operands are defined following a new field character, then the following defaults apply:

```
TYPE=OUTPUT INTENS=LOW
```

No special attributes or internal validation apply.

BLANKS={ TRAIL | NONE | ANY }

For input fields, this operand determines the format the entered data must take. By default, a field can contain embedded blanks (BLANKS=ANY). Specification of this operand helps ensure that the entered data does not contain embedded blanks, and contains only trailing blanks (TRAIL) or no blanks at all (NONE). This operand works independently of the REQUIRED operand. For optional fields, this operand can still be specified to help ensure that any data entered is in the correct format. If &CONTROL PANELRC is not in effect, BLANKS=TRAIL is specified, and the data is in error, Panel Services redisplays the panel with the &SYSMSG variable set to:

```
INVALID IMBEDDED BLANKS
```

If BLANKS=NONE is specified and the data is in error, Panel Services redisplays the panel with the &SYSMSG variable set to:

```
INCOMPLETE FIELD
```

If &CONTROL PANELRC is in effect, control is returned to the NCL procedure for error handling instead of being handled totally by Panel Services. In this case, &SYSFLD contains the name of the field in error and &SYSMSG the error message text.

CAPS={ YES | NO }

(Input fields only) Determines whether to convert entered data to uppercase before passing it to the NCL procedure in the nominated variable. Conversion to uppercase is also performed for the data associated with an input variable before displaying the panel. This does not affect the current contents of the variable, unless the data is modified and entered by the operator. Output fields are displayed exactly as defined and are not subject to uppercase conversion.

Uppercasing data for CAPS=YES fields uses the language code of the user region to select the character set that is used as the basis of the translation. Where the language code of the user is not one of the supported values, the language code of the system is used. Where the language code of the system is not one of the supported values, a translation based on EBCDIC codes is performed.

Note: Because data can be converted to uppercase before performing the assignment, the effect of CAPS=NO can be negated if the variable that receives the data is used in an assignment statement (for example, &A = &DATA) within the processing NCL procedure. See the &CONTROL UCASE option.

The CAPS operand is ignored when operating in a system executing with SYSPARMS DBCS=YES.

**{ COLOR | COLOUR }={ BLUE | RED | PINK | GREEN |
TURQUOISE | YELLOW | WHITE | DEFAULT}**

Applies only to IBM terminals with seven-color support and Fujitsu terminals with three- or seven-color support, and determines the color of the field.

If the terminal does not support extended color, the COLOR operand is ignored. This feature enables COLOR to be specified on panels that are displayed on both color and non-color terminals. COLOR can be used with the HLIGHT operand.

For Fujitsu terminals that support extended color data streams where only three colors are available, the following color relationships are used:

Specified	Result (on Fujitsu three-color terminals)
GREEN	GREEN
RED	RED
PINK	RED
BLUE	GREEN
TURQUOISE	GREEN
YELLOW	WHITE
WHITE	WHITE
DEFAULT	GREEN

Fujitsu seven-color terminals are treated the same as IBM seven-color terminals.

The DEFAULT keyword indicates that the color of the field is to be determined from the INTENS operand. This feature is useful if you want to set the color from an NCL procedure (that is, COLOR=&COLOR is specified and the NCL procedure can set the &COLOR variable to DEFAULT).

CSET={ ALT | DEFAULT }

(Output fields only) Determines which terminal character set to use to display the field. If you specify CSET=ALT (or ALTERNATE), you can draw boxes using the following characters:

e is displayed as │

s is displayed as ─

D is displayed as └

E is displayed a ┌

M is displayed as ┘

N is displayed as ┐

F is displayed as │

G is displayed as ⊥

O is displayed as ⊥ (rotated 90° counter-clockwise)

P is displayed as ⊥ (rotated 180°)

L is displayed as +

Note: CSET=ALT supersedes CSET=ASM in version 3.1

EDIT={ ALPHA | ALPHANUM | DATE*n* | DSN | HEX | NAME | NAME* | NUM | REAL | SIGNNUM | TIME*n* }

(Input field) Determines additional internal editing that Panel Services perform. By default no editing is performed. Specification of this operand helps ensure that the entered data conforms to the nominated type. If a field is mandatory, then also specify REQ=YES.

ALPHA

Accepts A to Z only.

ALPHANUM

Accepts A to Z, 0 through 9, #, @, and $ only.

DATE*n*

Field must be a valid date. The DATE*n* keyword must correspond to one of the &DATE*n* system variables, and the input field must contain date in the format associated with that system variable. For example, EDIT=DATE5 indicates that the input field must always contain a date in the format corresponding to the &DATE5 system variable (MM/DD/YY).

DSN

Specifies a valid OS/VS format data set name. If necessary, a partitioned data set (PDS) member name or Generation Data Group (GDG) number can be specified in brackets as part of the name.

HEX

Accepts 0 through 9 and A to F only.

NAME

Commences with alpha (A to Z, #, @, or $) and followed by alphanumerics (A to Z, 0 through 9, #, @, or $).

NAME*

Commences with alpha (A to Z, #, @, or $) and followed by alphanumerics (A to Z, 0 through 9, #, @, or $), but can be terminated with a single asterisk (*). This asterisk allows a value to be entered that can be interpreted as a generic request by the receiving procedure.

NUM

Accepts 0 through 9 only.

REAL

Input in this field must conform to the syntax for integers, signed numbers or real numbers, including scientific notation.

SIGNNUM

Field must be numeric but can have a leading sign (+ or -).

TIME*n*

Specifies a valid time. The TIME*n* keyword must correspond to one of the &ZTIME*n* system variables, and the input field must be in the format associated with that system variable.

When invalid data is detected and &CONTROL PANELRC is not in effect, Panel Services invokes standard error processing. Control is not returned to the NCL procedure until the error is corrected.

- For EDIT=NUM, the panel is redisplayed with the &SYSMSG variable set to:

  ```
  FIELD NOT NUMERIC
  ```

- For EDIT=REAL, the panel is redisplayed with the message:

  ```
  FIELD NOT REAL NUMBER
  ```

- For EDIT=ALPHA, ALPHANUM, HEX, or NAME, the panel is redisplayed with the &SYSMSG variable set to:

  ```
  INVALID VALUE
  ```

- For EDIT=DATE*n*, the panel is redisplayed with the &SYSMSG variable set to:

  ```
  INVALID DATE
  ```

- For EDIT=DSN, the panel is redisplayed with the &SYSMSG variable set to *one* of the following values:

  ```
  INVALID DATASET NAME
  ```

  ```
  INVALID MEMBER NAME
  ```

- For EDIT=TIME*n*, the panel is redisplayed with the &SYSMSG variable set to:

  ```
  INVALID TIME
  ```

In all cases, the terminal alarm is rung and the cursor is positioned to the field in error. If a #ERR statement has been included in the panel definition, processing of the error condition is performed as defined on that statement.

If &CONTROL PANELRC is in effect, control is returned to the NCL procedure for error handling instead of being handled totally by Panel Services. In this case, &SYSFLD contains the name of the field in error and &SYSMSG the error message text.

Note: Use of the EDIT operand might also require the use of the BLANKS operand to help ensure that entered data does not include embedded blanks. Regardless, editing is performed only for the length of the data entered and not for the length of the input field. If you want to enter the entire field, specify the BLANKS=NONE operand.

{ HLIGHT | HLITE }={ USCORE | REVERSE | BLINK | NONE }

Applies only to terminals with extended highlighting support, and determines the highlighting to be used for the field.

If the terminal does not support extended highlighting, the HLIGHT operand is ignored. This feature enables HLIGHT to be specified on panels that are displayed on terminals that do not support extended highlighting. HLIGHT can be used with the COLOR operand.

You can use NONE as a no-impact value when the highlighting of a field is being dynamically determined from the NCL procedure and set using variable substitution of the #FLD statement. When NONE is specified, the HLIGHT operand is ignored.

INTENS={ HIGH | LOW | NON }

Determines the intensity of the field when displayed.

HIGH

The field is displayed in double intensity. High intensity is typically associated with input fields and other important data. Minimize its use to maintain its effectiveness.

LOW

The field is displayed in low or standard intensity.

NON

The field is displayed in zero intensity, that is, any data within the field is not visible to the operator.

This operand is typically used for input fields for entering sensitive data such as passwords. Use of this attribute for output fields is meaningless. Color or extended highlighting attributes are ignored when used with this attribute.

JUST={ LEFT | RIGHT | ASIS | CENTER | CENTRE }

For output fields, this operand determines the alignment of the data within the field after trailing blanks have been stripped. Justification is applied at a field level. Do not confuse with VALIGN that applies to the individual variable only.

- JUST=LEFT results in padding to the right
- JUST=RIGHT results in padding to the left
- JUST=ASIS is treated as JUST=LEFT for output fields
- JUST=CENTER results in padding to both the left and the right.

For input fields, justification occurs both when the data is being displayed and when the data is being processed on subsequent entry. When an input field is formatted for display (the value currently assigned to the variable defined in the input field is substituted in place of the variable's name):

- The data is justified to the left and padded to the right for JUST=LEFT.
- The data is justified to the right and padded to the left for JUST=RIGHT.
- The data is aligned for JUST=CENTER as the data is aligned for JUST=LEFT.
- The data is positioned exactly as defined in the variable and padding to the right is performed for JUST=ASIS.

On subsequent reentry, trailing blanks and pad characters are stripped, unless the trailing pad character is a numeric, in which case it is not stripped:

- For JUST=RIGHT, leading blanks and pads are also stripped (including numerics). Use of JUST=RIGHT for input fields can inconvenience terminal operators because it is necessary to cursor across to the commencement of the data in the field.
- For JUST=ASIS, trailing blanks and pads are stripped, but leading blanks and pads remain intact.

MODE={ SBCS | MIXED }

Applies to IBM terminals capable of supporting DBCS data streams. If a panel is sent to such a device, input fields on the panel that use this #FLD character allow the operator to enter DBCS characters if MODE=MIXED is specified.

IBM DBCS terminals do not allow DBCS character entry in input fields that specify MODE=SBCS (single-byte character stream).

This operand does not apply to Fujitsu terminals, which allow DBCS character entry at any time.

NCLKEYWD={ YES | NO }

Specifies whether fields that use this FLD character accept input of words that conflict with NCL keywords. The default is YES. If you attempt to enter any NCL reserved keyword NO causes it to be rejected.

OUTLINE={ { L R T B } | BOX }

Specifies the extended highlighting outlining option required for this field. Any combination of L (left), R (right), T (top), or B (bottom) can be coded. The field is outlined at the top or bottom with a horizontal line and at the left or right border with a vertical line, according to the options specified. Alternatively, you can specify the BOX option, which is equivalent to specifying LRTB. This option is terminal-dependent.

PAD={ NULL | BLANK | *char* }

Applies to INPUT, OUTPUT, and SPD fields.

For output fields, PAD works with both the JUST and VALIGN operands, one of which must be specified for PAD to take effect. The operand determines the pad or fill character to use when the field is displayed.

The variable substitution process substitutes the data currently assigned to any variables within the field being processed. When substitution is complete, any difference between the length of the field defined on the panel and the length after substitution (after stripping trailing blanks) is padded with the specified PAD character.

NULL

Helps ensure that the terminal operator can use keyboard insert mode when entering data. Padding is performed either to the left or to the right, as specified in the JUST operand.

char

Specifies a single character that is to be the pad character (for example, PAD=-).

You can use any character, including the use of any of the field characters defined on #FLD statements. Take care when using numeric pad characters because their use affects the pad character stripping process on subsequent entry.

Using PAD characters with input fields invokes special processing on subsequent input to help ensure that unnecessary pad characters are stripped before returning the entered data in the nominated variable.

PSKIP={ NO | PMENU }

(Input fields only) Determines if panel skip requests are accepted in this field. A panel skip request is entered in an input field in the format =*m.m*, where *m.m* is a menu selection. When this request is entered in an appropriate field, a panel skip to the specified menu selection is performed.

NO

The input field is not scanned for panel skip requests.

PMENU

The input field is scanned for panel skip requests and action taken in response.

RANGE=(*min*,*max*)

(Numeric field) Specifies the range of acceptable values. The range includes all numbers, from the minimum number (*min*) to the maximum number (*max*). Both *min* and *max* must be specified, and max must be equal to or greater than min. Use of this operand forces EDIT=NUM. If the entered number falls outside the acceptable range and &CONTROL PANELRC is not in effect, Panel Services redisplays the panel, with the &SYSMSG variable set to:

```
NOT WITHIN RANGE
```

If &CONTROL PANELRC is in effect, control is returned to the NCL procedure for error handling instead of being handled totally by Panel Services. In this case, &SYSFLD contains the name of the field in error and &SYSMSG the error message text.

REQUIRED={ YES | NO }

Specifies whether a field is mandatory and the user must complete it. If &CONTROL PANELRC is not in effect, Panel Services rejects any entry by the user unless the mandatory field has been entered. If it is not entered, Panel Services redisplays the panel with the &SYSMSG variable set to:

```
REQUIRED FIELD OMITTED
```

The terminal alarm is rung and the cursor is positioned to the omitted field. If a #ERR statement has been included in the panel definition, processing of the error condition is performed as defined by the #ERR statement. Failure to include the &SYSMSG variable on the panel suppresses this error message. This operand can be abbreviated to REQ=.

If &CONTROL PANELRC is in effect, control is returned to the NCL procedure for error handling instead of being handled totally by Panel Services. In this case, &SYSFLD contains the name of the field in error and &SYSMSG contains the error message text.

SKIP={ YES | NO }

(Output field only) Determines whether to assign the skip attribute to the field. If the preceding input field is entered in full and the intervening output field is specified with the SKIP operand, this option causes the cursor to skip to the next input field.

This operand is NO by default, because field skipping can unexpectedly place the cursor in the wrong screen window when operating in split screen mode.

SUB={ YES | NO }

(Output field only) Determines whether to perform variable substitution. This operand is typically used only for fields where data contains the & character. Substitution results in the current value of that variable being substituted or, if no value is assigned, the variable being eliminated. This operand is ignored for both INPUT and SPD fields.

TYPE={ OUTPUT | INPUT | OUTVAR | SPD | NULL }

Determines whether to process the field as an output-only field (OUTPUT and OUTVAR), input field (INPUT), Selector Pen Detectable (SPD) field, or pseudo input field (NULL).

OUTPUT

A protected field is created, which does not allow keyboard entry. This field can contain a mixture of fixed data and variables. Each variable must commence with an ampersand (&). Substitution of variables is performed using the variables available to the invoking NCL procedure at the time the &PANEL statement is issued. Global variables can be referenced in an output field. Alignment and padding are performed according to the rules defined for the field.

INPUT

An unprotected field is created, which allows keyboard entry. This field must contain a single variable name (without the ampersand). This single variable must immediately follow the field character. System variables and global variables cannot be used in an input field. Subsequent data entered into this field is made available to the invoking NCL procedure in this variable on return from the &PANEL statement. Specification of multiple variables or a mixture of variables and fixed data in an input field results in an error.

OUTVAR

Is the same as TYPE=OUTPUT, except that Panel Services inserts an & between the field attribute and the next character. This feature means that you can follow a TYPE=OUTVAR field character with a variable name without the ampersand. This facility makes it easy to create fields which switch between input and output under NCL control. For example, a panel contains the statements:

```
#FLD $ TYPE=&INOUT
+ Record Key .....$RKEY +
```

An NCL procedure would then set the variable &INOUT to control whether the data in the variable &RKEY is output only or an operator can modify it:

```
&INOUT = OUTVAR -* the value is output only.
&INOUT = INPUT  -* the Operator can modify the
                -* field.
```

A similar effect can be achieved using &ASSIGN OPT=SETOUT.

SPD

A protected field is created in selector pen detectable format. This value enables the terminal operator to select the field using either a LIGHT PEN or the CURSOR SELECT key. The SPD field characters must be immediately followed by one of the three designator characters (?, &, or a blank), which can in turn optionally be followed by one or more blanks. A single variable with no other fixed data must also be defined within the field. This single variable must be defined without the ampersand (&) and cannot be a system or global variable. If selected by the user, the variable nominated in the SPD field is set to the value SELECTED on return to the NCL procedure. If not selected, the variable is set to a null value.

NULL

An unprotected field is created, which allows keyboard entry. However, the name of an input variable to receive data entered in the field is not required because any data entered by the terminal operator in a TYPE=NULL field is ignored. Display data in this field can be in any format. The NULL option accommodates four-color terminals where the field attribute byte is used to determine the color in which the field is displayed. (Seven-color terminals use an extended data stream to set the color). The NULL option indicates that Panel Services is to use an unprotected field attribute with the INTENS operand value to determine the color of the field.

VALIGN={ NO | LEFT | RIGHT | CENTER | CENTRE }

(Output fields only) Determines the alignment of data for an individual variable only. Do not confuse with the JUST operand, which applies to field alignment after all variable substitution has been completed. The VALIGN operand is designed to facilitate tabular output without the need to specify many individual field characters on the panel. If specified for an input field, the operand is ignored.

The substitution process normally substitutes the data assigned to a variable in place of the variable name. No additional blanks are created or removed during this process. Thus, if the data being substituted is shorter than the name of the variable (for example, variable &OUTPUTDATA set to 5678), then data following the variable name is shifted left to occupy the area remaining after the removal of the variable name. This shift would destroy any tabular alignment where the length of the data for each variable differed. If the data being substituted is shorter than the variable name, the VALIGN option helps ensure that the data to the right of the variable is not shifted to the left. The length of the variable name (including the ampersand) is the important factor and determines the number of character positions to preserve during the substitution process.

However, data is not truncated and, if the data being substituted is longer than the variable name, then the data to the right is moved to accommodate all the substituted data. VALIGN works with the pad character specified on the PAD operand. The pad character is used to fill any difference between the data being substituted and the length of the variable name being replaced.

VALIGN=NO

No alignment or padding is performed.

VALIGN=LEFT

Data is aligned to the left and padded to the right. An abbreviation of L is acceptable.

VALIGN=RIGHT

Data is aligned to the right and padded to the left. An abbreviation of R is acceptable.

VALIGN=CENTER

Data is centered (or one position to the left for an odd number of characters) and padded both to the left and to the right. An abbreviation of C is acceptable.

Examples: #FLD Control Statement

```
#FLD # TYPE=INPUT REQ=YES EDIT=NUM COLOR=RED RANGE=(1,3)
#FLD # BLANKS=TRAIL PAD=_
#FLD # TYPE=OUTPUT COLOR=&COLOR HLIGHT=&HLIGHT
#FLD ( TYPE=INPUT INTENS=HIGH EDIT=DATE4
#FLD @ HLIGHT=BLINK
#FLD / TYPE=SPD
#FLD % JUST=R PAD=- -* supplementing default output char
#FLD _ JUST=ASIS -* supplementing default input char
#FLD + VALIGN=RIGHT JUST=CENTER -* null pad assumed
```

Notes:

- If insufficient space is available on a single statement, multiple #FLD statements can be used for the same field character.
- Symbolic variables can be included in a #FLD statement. Variable substitution is performed before the statement is processed, using variables available to the NCL procedure at the time the &PANEL statement is issued.
- You can alter the default field characters by using the DEFAULT operand of the #OPT control statement.

- You can use the #ERR control statement to simplify redisplaying a panel to indicate a field in error.
- The &CONTROL PANELRC operand can be used to specify that the NCL procedure receives control for further processing when internal validation detects an error in data entered by the operator. When this technique is used, the procedure can determine the field in error (from the &SYSFLD variable) and the error message to issue (from the &SYSMSG variable). With this information, it can alter its processing accordingly, including altering the text of the error message in the &SYSMSG variable if necessary.

#NOTE Control Statement—Allow User Comments in a Panel Definition

This control statement provides a means of placing documentation within a panel definition. The #NOTE statement is not processed and is ignored. Multiple #NOTE statements can be specified. However, as with the #FLD, #ERR, and #OPT statements, all #NOTE statements must precede the start of the panel. The start of a panel is determined by the first line that is not a control statement or #NOTE statement.

This control statement has the following format:

#NOTE *any_text*

any_text

Specifies any free-form user text.

Examples: #NOTE Control Statement

```
#NOTE This panel is used by the Network Error Log System
#NOTE INWAIT=60 CURSOR=&CURSORFLD
```

As shown in the example, the #NOTE statement can provide a simple means of temporarily nullifying another control statement, allowing for easy reinstatement when required.

#OPT Control Statement—Define Panel Processing Options

This control statement tailors the processing options for a panel.

Before parsing, the #OPT statement is scanned and any variables are substituted. This process makes it possible to tailor dynamically any of the operands on the statement.

Multiple #OPT statements can be specified. However, as with #FLD, #ERR, and #NOTE statements, all #OPT statements must precede the start of the panel, as determined by the first line that is not a control statement.

This control statement has the following format:

```
#OPT [ ALARM={ YES | NO } ]
     [ BCAST={ YES | NO } ]
     [ CURSOR={ varname | row,column } ]
     [ DEFAULT={ hlu | X'xxxxxx' } ]
     [ ERRFLD=varname ]
     [ FMTINPUT={ YES | NO } ]
     [ IPANULL={ YES | NO } ]
     [ INWAIT=ss.th ]
     [ LSM={ YES | NO } ]
     [ PREPARSE={ (c,S) | (c,D) } ]
     [ UNLOCK={ YES | NO } ]
     [ MAXWIDTH={ YES | NO } ]
```

ALARM={ YES | NO }

Determines whether to ring the terminal alarm when the panel is displayed. Dynamic control of the alarm can be achieved by changing the value of the ALARM operand using a variable set before issuing the &PANEL statement.

If internal validation has detected an error and the panel is being redisplayed to indicate the error, this operand is ignored and the terminal alarm rung. The #ERR statement can be used to alter the processing performed when an error condition is detected.

BCAST={ YES | NO }

Specifies whether to redisplay the panel automatically when a broadcast is scheduled. By default, the only panels that are redisplayed automatically are those panels that contain one or more of the special broadcast variables, &BROLINE*n*. If BCAST=YES is coded, a broadcast redisplays the panel even if it does not contain any of the &BROLINE*n* variables.

CURSOR={ *varname* | *row,column* }

Specifies the name of a variable in either an INPUT or an SPD field where the cursor is to position. Alternatively, the precise coordinates for the cursor can be defined (see page 124) as *row*,*column*.

The value of *varname* is the variable name without the ampersand, as used in the INPUT or SPD field (for example, CURSOR=FIELD5).

Where coordinates are specified, row must be specified in the range 1 through 62 and column in the range 1 through 80. The row and column values are always relative to the start of the current window and therefore remain unchanged when operating in split screen mode. The &CURSROW and &CURSCOL system variables can be used to determine the location of the cursor on input to the system.

Dynamic positioning of the cursor can be achieved by using a variable or variables (including the ampersand) in place of *varname* or *row*,*column*. The invoking NCL procedure can set the variables to the name of the field to contain the cursor or the coordinates before issuing the &PANEL statement.

If internal validation has detected an error and the panel is being redisplayed to indicate the error, the CURSOR operand is ignored and the cursor is positioned to the field in error.

Specifying *varname* with a name other than the name of a variable used in an INPUT or SPD field results in an error. If coordinates are used and they lie outside the dimensions of the window currently displayed, the cursor is positioned in the upper left corner of the window.

DEFAULT={ *hlu* | X'*xxxxxx*' }

Alters the three standard default field characters. If the #OPT statement is omitted or the DEFAULT operand not used, then three standard field characters are provided for use when defining the panel. They are:

% = protected, high-intensity

+ = protected, low-intensity

_ = unprotected, high-intensity

It is sometimes necessary to select alternative field characters, for example, if the underscore character is required within the body of the panel for some reason.

The DEFAULT=*hlu* operand must always specify three characters. The characters chosen must *not* be alphanumeric, that is, any special character except ampersand (&), which is reserved for variables. They must not duplicate another field character, except one already defined as a default. The order of the characters is significant, as the attributes of the standard default characters apply in the order described.

Therefore specification of DEFAULT=*+/ results in:

* = protected, high-intensity

+ = protected, low-intensity

/ = unprotected, high-intensity

You can also specify the default field characters in hexadecimal in the format:

`DEFAULT=X'`*`xxxxxx`*`'`

Each *xx* pair represents a hexadecimal number in the range X'00' to X'FF'. All numbers except X'00” (null), X'40' (blank), and X'50' (ampersand), are valid. This format even allows alphanumeric characters to be used as field characters. For example, if you specify X'C1', any occurrence of the letter A in the panel definition is treated as the default character. However, we recommend using hexadecimal values that do not correspond to alphanumeric characters.

For example, specification of DEFAULT='010203' would result in:

X'01' = protected, high-intensity

X'02' = protected, low-intensity

X'03' = unprotected, high-intensity

ERRFLD=*varname*

Specifies the name of a variable in an INPUT field that is in error and for which to invoke error processing, as defined on a #ERR statement. Use of this operand without including a #ERR statement within the panel definition results in an error. The ERRFLD operand provides a simple way of informing Panel Services that the field identified by *varname* is in error. Panel Services displays the panel using the options defined on the #ERR statement. The #ERR statement could indicate to display the error field in reverse video, colored red and the terminal alarm rung. The assignment of error text into a variable appearing on the screen to identify the nature of the error typically accompanies the use of the ERRFLD operand.

This operand is typically specified with the name of a variable (including the &) that is set to null unless an error occurs. On error, the NCL procedure sets the variable to the name of the field in error before issuing the &PANEL statement to display the panel.

ERRFLD provides the panel designer with a simple means of changing the attributes of a field (such as color and highlighting) without needing to resort to dynamic substitution of #FLD statements.

Consider the case where an input field &INPUT1 is found to be in error and the #OPT statement has been defined with ERRFLD=&INERROR. The NCL procedure simply assigns the name of the variable used to identify the input field, in this case INPUT1 (minus the &), into &INERROR and then redisplays the panel.

```
&INERROR = INPUT1
&SYSMSG = &STR THIS FIELD IS WRONG
&PANEL MYPANEL
```

In this example, the text that identifies the nature of the error has been assigned into the variable &SYSMSG which would be defined somewhere on the panel.

&ASSIGN OPT=SETERR is effective only if &CONTROL FLDCTL is in effect.

FMTINPUT={ YES | NO }

Determines whether input fields are to be formatted when a panel is displayed. This specialized option is designed to be used with INWAIT. When processing with INWAIT, the time interval could expire at the instant when the operator enters data. If the same panel is redisplayed to update the screen contents, the data entered by the operator is lost as the new panel is written. FMTINPUT can be used to bypass formatting of input fields and hence when the panel is redisplayed only output fields are written. The value of this operand is typically assigned to a variable from within the NCL procedure, and changed between YES and NO as required (#OPT FMTINPUT =&YESNO). Take care when using this facility because incorrect use of FMTINPUT=NO can result in validation errors. Ideally, a panel is displayed initially with FMTINPUT=YES and only when the INWAIT timer expires would it be redisplayed with FMTINPUT=NO.

IPANULL={ YES | NO }

Specifies whether to set all variables associated with the panel input fields to null when the panel is displayed with the INWAIT option, and the time specified on the INWAIT expires so that control is returned to the procedure without any panel input or a PA key causes the input.

If you do not want to erase input field variables if INWAIT completes or a PA key is pressed, specify IPANULL=NO.

Default: YES

INWAIT=*ss.th*

Specifies the time in seconds and parts of seconds that Panel Services is to wait for input from the terminal before returning control to the NCL procedure following the &PANEL statement. By default, the system, having displayed a panel, waits indefinitely for input. This indefinite wait is not always desirable, as is the case where a terminal is performing a monitoring function where input can be infrequent or never occur. If INWAIT is utilized and the specified time elapses, control is returned to the NCL procedure with all input or SPD variables set to null. If input is made during the time interval, the time period is canceled and standard processing proceeds.

The maximum value that can be specified for INWAIT is 86400.00 seconds (24 hours).

Specification of part seconds is possible. For example:

```
INWAIT=.5
INWAIT=20.5
INWAIT=.75
```

Any redisplay of a panel (for example, by use of the clear key) causes the time interval to be reset. If the time interval expires before input is received, specification of internal validation options (such as REQUIRED=YES) is ignored.

Specification of INWAIT=0 or INWAIT=0.00 indicates to accept no input, and control is returned to the NCL procedure immediately after the panel has been displayed. In this case, subsequent action taken by the procedure determines the period that the panel remains displayed.

The invoking NCL procedure can determine, by testing the &INKEY system variable, whether the INWAIT time elapsed or data was entered. &INKEY is typically set to the character value of the key pressed by the operator to enter the data (for example, Enter or F1). If the INWAIT time interval elapsed and no entry was made, the &INKEY variable is set to null. If processing with &CONTROL PANELRC in effect, &RETCODE is set to 12 to indicate that the INWAIT timer has expired.

INWAIT is ignored for asynchronous panels.

Default: Wait indefinitely for input.

LSM={ YES | NO }

If you do not want the LSM to control a particular panel, then code #OPT LSM=NO in the panel definition. The entire panel is then written to the terminal each time. If large numbers (thousands) of EASINET terminals are being supported, the reduction in storage can become significant.

PREPARSE={ (*c*,S) | (*c*,D) }

Preparsing provides a means for dynamically modifying the location of field characters in a panel. Normally, the position of field characters (as defined by the #FLD control statement) is determined when Panel Services creates the panel and remains fixed until the panel is modified.

Although you can modify the attributes of each field character (such as the color of the field) by using variables in the #FLD statement, you are limited in the number of variations.

The PREPARSE operand requests that Panel Services performs a preliminary substitution scan of each panel line before processing the line for field characters. The PREPARSE operand specifies a substitution character (*c*) that is used to determine where substitution takes place. This character is processed in the same manner as an ampersand (&) is processed during standard substitution.

The ability to specify a character other than an ampersand means that preparsing does not affect standard substitution when it is performed following preparsing. You can use preparsing for the following purposes:

- Alter a field character that appears in a particular position to allocate a new set of attributes to the field.
- Create entire new fields (or complete lines) that in themselves contain the required field characters.

(*c*,S)

Indicates that the character *c* is used as the preparse character for the panel, but that the Static Preparse Option applies during preparse processing. This option prevents the movement of preparse or field characters during the substitution process. This option is useful when panels are being dynamically modified to hold data that can vary in length but displays in columns. Substituted data can be truncated if it is too long to fit into its target field without overwriting the next occurrence of a preparse or field character on the same line.

(*c*,D)

Indicates that the character *c* is used as the preparse character for the panel, but that the Dynamic Preparse Option applies during preparse processing. The dynamic option allows the movement of preparse or field characters to the left or right of their original position to accommodate differing lengths of data being substituted into the panel.

UNLOCK={ YES | NO }

Determines whether the terminal keyboard is unlocked when the panel is displayed. Specification of UNLOCK=NO prevents entry of data by the terminal operator. NO is typically used with the INWAIT operand where a panel is being displayed for a short period before progressing to some other function.

MAXWIDTH={ YES | NO }

When MAXWIDTH=NO is specified or defaulted, the panel display is limited to the standard 80-column width. If you are designing a panel for a wider screen (for example, a model 5 terminal), specify MAXWIDTH=YES to allow the full width of the screen to be used.

Examples: #OPT Control Statement

```
#OPT DEFAULT=#$%
#OPT INWAIT=60 CURSOR=&CURSORFLD
#OPT CURSOR=IN1 ALARM=YES
#OPT ALARM=&ALARM PREPARSE=($,D)
#OPT ERRFLD=&INERROR
#OPT INWAIT=.5 UNLOCK=NO PREPARSE=($,S)
#OPT CURSOR=5,75
#OPT CURSOR=&ROW,&COLUMN FMTINPUT=&FMT
```

Notes:

- Multiple #OPT statements can be used if necessary.
- Symbolic variables can be included in a #OPT statement. Variable substitution is then performed before the statement is processed.
- A panel is redisplayed automatically following use of the CLEAR key. Control is not returned to the invoking NCL procedure.
- The attributes of the standard default characters can be modified using a #FLD statement that adds additional attributes (such as color) or alters existing attributes.

#TRAILER Control Statement—Place Lines at Screen End

This control statement positions function key prompts at the bottom of the screen.

Indicate the start of the trailer lines with a #TRAILER START statement. Then enter the lines to appear at the end of the screen, followed by a #TRAILER END statement.

This control statement has the following format:

```
#TRAILER [ START | END ]
         [ POSITION={ YES | NO } ]
```

START

Specifies the start of the lines in the trailer. Each line following this line until a #TRAILER END statement or another control statement such as #FLD is placed in the trailer.

END

Specifies that the end of the lines in the trailer. There must have been a #TRAILER START statement earlier in the panel definition.

No other operands can be specified on a #TRAILER END statement.

POSITION={ YES | NO }

Specifies whether to display the trailer lines.

YES

The trailer lines are displayed on the final lines of the physical screen.

NO

This value can be used to suppress the display of the trailer lines, even though they remain in the panel definition.

Examples: #TRAILER Control Statement

```
#TRAILER START
%This appears on the last line of the panel
#TRAILER END
```

Notes:

- The #TRAILER statements must appear before the first panel line in the definition. If you want to preparse the lines, you place the #TRAILER statements after the #OPT PREPARSE= statement.
- The field attribute characters which you use in the trailer lines can be defined before or after the trailer lines in the panel.
- The trailer lines cover any panel lines that would otherwise have been displayed.
- The trailer lines are positioned so that they end at the bottom of the physical screen if the window starts at the top of the screen.

Chapter 8: NCL File Processing

This section contains the following topics:

UDB File Formats

The term UDB refers to any file processed with the NCL file processing statements. Three types of physical file come under this heading:

- Mapped format files
- Unmapped format files
- Delimited format (or UDB format) files

Mapped Format Files

Mapped format processing can be used on any file which is arranged in a manner that is describable to Mapping Services using a map. A map is used to describe a file in terms of structures or components understood by Mapping Services.

Mapped format file access is the preferred method of file processing in NCL. The advantages of mapped format processing are:

- Allows for transparent data to be placed in the file
- Allows for upward compatibility
- Combined with Mapping Services, allows NCL to operate at a logical level, even when dealing with complex data formats

Default Map

A default mapping, using the $NCL map can be used to maintain NCL variable data in a file. Individual data records up to 32K and containing transparent variable data, are supported in this manner. Each variable exists in a vector format and can be isolated using length fields and data tags.

When using the $NCL map, the file statements reference one or more NCL variables in the normal manner. The actual structure of the record can be understood by referring to the distributed $NCL map.

Other Maps

When a retrieval is being carried out from a file using a mapped format, the contents of the file records are read into a mapped data object (MDO). A map is attached to the MDO to provide NCL with a means of interpreting its contents. The &ASSIGN verb can then be used to reference certain sections of the data by name and extract these from the MDO into NCL tokens (variables). Mapping Services does the work of locating the sections of data which are being extracted by using the map. By doing the work of locating components within the data, Mapping Services can save a considerable number of NCL statements, particularly for files that have a complicated format, and also for files that contain non-printable data.

Because the map definition is a separate entity within your product, it is also possible that subsequent changes to the file format will only require changes to be made to the map definition, and not to the NCL code itself.

Mapping Services can be used to define maps capable of interpreting most data formats.

Unmapped Format Files

NCL can also be used to process records from VSAM files for which no map is available.

Unmapped mode processing is generally used in applications where UDBs created or used by NCL procedures are to be processed offline by other systems.

NCL file processing statements allow the NCL user to indicate that a particular UDB is to be processed in unmapped mode, causing NCL to bypass any attempt to identify individual fields within records read from or written to the file.

While still allowing all the simplicity of access to the file, unmapped mode processing means the NCL user must know the structure of records on the file. Consequently more attention must be paid to this aspect of file processing than is the case for the simpler delimited format files.

Print (Using Unmapped Format File Support)

Unmapped format is for processing data of any format including binary or non-printable data. It is also useful for report generation where the data is to be directed to a system printer. On a z/OS system, the unmapped format is ideal for generating reports that are to be routed to JES SYSOUT for printing either on local or remote printers. The NCL provides operands that help control report formatting.

For more information about the &FILE ADD statement, see the *Network Control Language Reference Guide*.

If required, a mapped format or delimited format file can be processed in unmapped mode.

The processing mode (UDB format, mapped, or unmapped) for any file can be swapped at any time during NCL procedure processing and this might become necessary if special key structures are being used.

Delimited Format Files

Data in UDB format files is stored in records that contain any number of fields to a maximum total record size of 32K, which is the maximum statement size that can be processed by NCL. Each field is isolated from the next by a high-value (X'FF') field separator and the last field in a record is followed by a field separator. The length of a field is determined by the length of data supplied by the NCL user. The occurrence of two consecutive field separators indicates the presence of a null field that contained no data when the record was created, even though the field concerned is logically part of the record. The total length of any record includes the key and all field separators.

The rules for field separators within UDB format files are:

- A field separator follows each field
- No field separator follows the key
- A trailing field separator follows the last field, unless it is a key-only file with no data other than the key
- Two consecutive field separators indicate a field of 0 length (a null field)

Multiple File and Alternate Index Support

NCL supports the concurrent processing of multiple files (including a mixture of delimited format, mapped format, and unmapped format files) in addition to the use of VSAM alternate indexes that allow records to be retrieved using keys based on different parts of the data. For example, a UDB used to maintain Help Desk problem information might contain a sequentially allocated problem number, in addition to the name of the resource to which the problem was related. Using alternate indexes, it is possible to retrieve records in both problem number order and resource name order.

Work with UDBs

UDB support is a standard function of NCL. NetMaster Databases (NDBs) provide more advanced facilities for users with requirements for more complex, high-performance data management within their NCL procedures.

NCL supports the following techniques for VSAM user database (UDB) files:

- Processing by specific key (KSDS)
- Processing by partial or generic key (KSDS)
- Sequential processing (KSDS), in ascending or descending key order
- Retrieval for update or deletion (KSDS)
- Deletion by specific key (KSDS)
- Multiple record deletion by generic key (KSDS)
- Sequential processing, forwards or backwards (ESDS)
- Emptying (resetting) of data sets (KSDS)
- Processing of SYSOUT data sets (z/OS only)

NCL uses the VSAM access method for its speed and flexibility. The high-level NCL implementation shields the user from all VSAM complexities.

UDBs are keyed VSAM data sets (KSDS) or sequential VSAM data sets (ESDS). For keyed data sets, any key length from 1 to 255 characters can be selected. Standard VSAM restrictions regarding duplicate keys apply.

Allocate UDBs

You can allocate UDBs dynamically by using the ALLOCATE command.

Note: For more information about the ALLOCATE command, see the online help.

The UDBs allocated in this manner must be identified to your product as UDBs through use of the UDBCTL command as follows:

```
UDBCTL OPEN=ddname ID=file_id options
```

ddname

Specifies the data definition (DD) name of the DD card defining the UDB.

file-id

Specifies a logical file ID. You can specify ID=* to use the same value for both *ddname* and *file_id*.

options

Specifies any special processing attributes to apply to this UDB.

Dynamically allocated UDBs can be deallocated (using the DEALLOCATE command) after the UDBCTL command has been used to close them.

Prepare to Use a UDB

The process to create and use a UDB is:

- Determine the format of the data set, the length of the records it is to contain and the size of the key to be used. If records are to be added or updated on-line, the key must commence in the first position of the record (unless the data set is to be processed in unmapped mode). The amount of space required should also be determined at this stage. Remember to take the presence of field separators into account when deciding on record length. Normally field separators have little effect on overall record length, but they can become significant when a record contains numerous very short fields.
- Use the IDCAMS VSAM utility to define the data set. The DEFVSAM member in the distribution library contains an example of defining a VSAM data set.
- Allocate the file dynamically with the ALLOCATE command.

- Use the UDBCTL command to open the UDB and assign a logical file ID, and any special processing attributes that might be required. This makes the file available for processing.
- Include the appropriate &FILE OPEN statement in the NCL procedure.

After these steps have been performed, the file is available for processing using standard NCL statements.

UDB Initialization

Your product attempts to open UDBs when a UDBCTL OPEN command is processed. If the open fails, the system internally uses IDCAMS to verify the data set and then retry the open. Therefore it is not necessary to include verification steps in the system JCL.

Initialization of KSDS UDBs

If the data set requires an initial load, the system loads a single record with a key of all X'00' and then deletes it. If this load fails, the data set is closed and is classified as unusable, and is blocked from further processing until the problem has been rectified. Use the SHOW UDB command to determine the cause of the error.

Initialization of ESDS UDBs

Empty ESDSs identified as UDBs are initialized by loading a single record which has the format:

```
N28510 VSAM INITIAL LOAD PERFORMED AT hh.mm.ss ON day-dd.mon-year
```

This record is always the first record of an ESDS UDB, unless the ESDS already contains records when opened by your product. NCL procedures that reference the UDB should ignore this first record. If preparing for unmapped processing of an NCL-created ESDS UDB, use IDCAMS to REPRO the data set skipping, the first record.

Write to SYSOUT as an ESDS

An ESDS UDB that is actually a SYSOUT data set (z/OS systems only), allocated dynamically using the ALLOCATE command, does not require initialization and therefore this processing is bypassed. SYSOUT data sets can therefore be written directly from NCL procedures without an initialization record appearing on the output.

Notes:

- Your product does not load alternate indexes. You must do this using the IDCAMS BLDINDEX function.
- The Dataset Services SUBMIT option provides a simpler way of creating SYSOUT data sets.
  For more information about Dataset Services options, see the *Network Control Language Reference Guide*.

Control UDB Performance and Resource Usage

The techniques used by NCL should ensure efficient processing of VSAM files. Additional performance gains can be obtained by the allocation of additional buffers and processing strings. This is achieved using the JCL AMP statement sub parameters on the DD statement for the file or options on the UDBCTL command:

BUFNI

The number of index buffers to be allocated by VSAM

BUFND

The number of data buffers to be allocated by VSAM

STRNO

The maximum number of concurrent strings VSAM is to use

By default, your product allocates 2 data buffers, 3 index buffers and 2 processing strings unless alternative values are provided as described.

This buffer allocation applies per string-that is, allocation of a complete set of buffers is performed by VSAM for each concurrent position held on the UDB. Where the usage of a UDB is such that a large number of concurrent accesses to the UDB might be possible, care must be taken that VSAM buffer allocations do not lead to storage shortages affecting the performance of other product functions. Also, in cases such as these, NCL procedures should be written to avoid the maintenance of generic UDB processing environments over long periods.

While varying these parameters can offer significant performance benefits, they should only be changed if the impact on VSAM processing is clearly understood. Incorrect changes can impose severe storage overheads which could impact the operation of other system components.

Add Records to a UDB

The &FILE ADD statement is used to add new records to a UDB. The &FILE PUT statement can also be used to add records. However, whereas &FILE ADD receives an error indication if a record with a like key exists, &FILE PUT will replace a record with a like key. If an NCL procedure is known to be creating new records then &FILE ADD should be used.

Before a record can be added, the key of the record must be identified. This is done using the KEY= or the KEYVAR= operand on the &FILE statement. The KEY= operand can be used in conjunction with the ADD and PUT operands.

The &FILE ADD statement is used to supply the data of the record to be added to the UDB. Depending on the format of the UDB this could be a text string, tokens, or an MDO.

The success of the &FILE ADD statement can be tested using the &FILERC (file return code) system variable. If the &FILE ADD statement was not successful, the &VSAMFDBK system variable will contain a standard VSAM completion code indicating the type of error that occurred. For example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE ADD ID=MYFILE KEY='RECORD1' VARS=A* RANGE=(1,7)
&IF &FILERC NE 0 &THEN &WRITE ERROR CODE=&VSAMFDBK
```

SYSOUT Considerations

A SYSOUT data set (z/OS) can be defined as a UDB and be the subject of &FILE ADD (or &FILE PUT) statements. The SYSOUT data set appears to your product as a VSAM ESDS.

NCL procedures can use &FILE ADD or &FILE PUT statements to write to SYSOUT UDBs. The UDB should be treated as a mapped or unmapped format file and processed using the unmapped format option.

NCL supports both ANSI and machine print control characters. However, CA recommends that you use ANSI characters because they are easier to use.

On z/OS systems, NCL determines the record format (RECFM) of a SYSOUT data set to decide if the data set is to be supported with ANSI print control characters (RECFM=A) or machine control characters (RECFM=M). When the SYSOUT data set is dynamically allocated the PRTCNTL operand can be used to specify which format is required.

Format SYSOUT Output

When processing SYSOUT data sets in ANSI format the &FILE PUT/ADD statement supports specification of formatting options such as skipping to new pages, line spacing, bolding, and underscoring using the PRTCNTL operand. As such the print character is never formatted by the user as part of the output data. When processing with machine control characters, the user is responsible for formatting the output data with the appropriate control character as the first character of the output data.

When processing an ANSI format SYSOUT file (RECFM=A or PRTCNTL=A) additional sub-operands of PRTCNTL allow left or right justification or centering of the data. In such cases the logical width of the report is determined by the logical record length (LRECL) of the data set. This too can be specified on the ALLOC command using the LRECL operand. The width specified does not include the print control character which will be allowed for and added internally.

If the data set is the z/OS internal reader (INTRDR, which must be allocated dynamically), it can be treated as an unmapped format UDB, in which case written records are interpreted as JCL. In this way, jobs can be submitted for execution.

When submitting jobs in this way, the job number of the submitted JCL stream can be obtained by issuing &FILE GET ID=name END. The job number is returned in the variable &ZJOBNUM.

Update Records in a UDB

The &FILE PUT statement is also used to update records in a UDB. A record can either be updated by first reading it with an &FILE GET statement and then replacing it with an &FILE PUT statement or by replacing it directly with an &FILE PUT statement. The method chosen will be determined by the application. Regardless of the method to be utilized the key of the record is identified using the KEY= or KEYVAR= operand on the &FILE statement.

If the key was specified on the &FILE GET statement, there is no need to specify it again on the &FILE PUT statement because the retrieved key remains current for the file.

If the possibility exists that multiple updates for the same record can be attempted concurrently, precautions must be taken to ensure that updates are not lost or inadvertently overwritten.

The success of the &FILE PUT statement can be tested using the &FILERC system variable. If the &FILE PUT statement is not successful the &VSAMFDBK system variable contains a standard VSAM completion code indicating the type of error that occurred. For example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE GET ID=MYFILE KEY='RECORD1' VARS=B*
&IF &FILERC NE 0 &GOTO .NORECORD
     .
     .
     .  update data
     .
     .
&FILE PUT ID=MYFILE ARGS RANGE=(1,7)
&IF &FILERC NE 0 &WRITE DATA=ERROR CODE=&VSAMFDBK
```

&FILE PUT will replace a record which already exists. Where multiple concurrent updates are possible the NCL procedure must ensure that it has exclusive control of a record prior to issuing the &FILE PUT statement to update it. This can be achieved by first reading the record with an &FILE GET statement that specifies the OPT=UPD operand. This operand indicates that exclusive control of the record is required. If the record is already in use, the &FILE GET statement fails with &FILERC set to 8 and &VSAMFDBK set to 14. The NCL procedure can then take alternative action, such as delaying processing for a short period before retrying the &FILE GET. Having obtained the record the &FILE PUT statement can be issued to complete the update.

When using &FILE GET with the UPD option, the NCL procedure should complete processing of the record as quickly as possible. It is not good practice to obtain exclusive control of a record and then issue a full-screen panel which waits for operator input. This can delay other users for excessive periods.

An example of a procedure using the &FILE GET ... OPT=UPD operand follows. This example issues a one second delay if exclusive control of the record cannot be obtained, and then retries the &FILE GET.

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
.RETRY
   &FILE GET ID=MYFILE KEY='RECORD1' OPT=UPD ARGS
   &IF &FILERC EQ 0 &THEN &GOTO .UPDATE
   &IF &FILERC EQ 8 AND &VSAMFDBK EQ 14 &THEN &GOTO .WAIT
   &ENDAFTER &WRITE DATA=ERROR CODE=&VSAMFDBK

.WAIT
   &DELAY 1
   &GOTO .RETRY
.UPDATE
   .
   .
   .  update data
   .
   .
&FILE PUT ID=MYFILE ARGS RANGE=(1,7)
&IF &FILERC NE 0 &THEN &WRITE DATA=ERROR CODE=&VSAMFDBK
```

Delete Records from a UDB

The &FILE DEL statement is used to delete records from a UDB. Two techniques can be used:

- Deletion by specific key-the full key of the record to be deleted is supplied. Only the record with this key is deleted.
- Deletion by generic key-a partial key is supplied. Any record that matches this partial key (KEQALL) or is equal to or greater than this partial key (KGEALL) is deleted.

The KEY= or KEYVAR= operand can be used to specify the full or partial key to be used for the delete.

Deletion of a single record can be achieved as shown in the following example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE DEL ID=MYFILE KEY='RECORD1'
```

If a partial key is provided for a single record delete, NCL deletes the first record on the file that matches the partial key.

If a group of records is to be deleted, a partial key that identifies the first record in the group must be specified on the KEY= or KEYVAR= operand. Then either of the following two values must be set for the OPT= operand.

KEQALL

Delete this record and all following records that have keys equal to the partial key provided.

KGEALL

Delete this record and all following records that have keys equal to or greater than the partial key provided.

The success of the &FILE DEL statement can be tested using the &FILERC system variable. If the &FILE DEL statement was not successful, the &VSAMFDBK system variable will contain a standard VSAM completion code indicating the type of error that occurred. For example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE DEL ID=MYFILE KEY='RECORD' OPT=KEQALL
&IF &FILERC NE 0 &WRITE DATA=ERROR CODE=&VSAMFDBK
```

The number of records deleted by an &FILE DEL statement is returned in the system variable &FILERCNT, which can then be used, for example, as feedback information for display on a panel. That is:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE DEL ID=MYFILE KEY='RECORD' OPT=KGEALL
&WRITE &FILERCNT RECORDS DELETED
```

&FILERCNT remains until the next &FILE DEL, or the file is closed, or processing changes to a different file. In the last case, &FILERCNT changes to reflect the number of records deleted on the file that is now being processed.

Use of generic delete functions is of value when large numbers of records are to be deleted from a UDB and gives better performance than the use of multiple single record deletes.

Retrieve Records from a UDB

The &FILE GET statement is used to retrieve records from a UDB. A number of techniques can be used:

- Retrieval by specific key-the full key of the required record is supplied. Only a record with this key will be returned.
- Retrieval by generic key-a partial key is supplied. Several values can then be used on the OPT= operand to determine the first record retrieved and the direction in which processing will continue: KEQ, KGE and KGT return the first record with a key equal to, equal to or greater than, or greater than the partial key (respectively), and set the retrieval direction to forwards, while KEL, KLE and KLT return the highest record with a key equal to, equal to or less than, or less than the partial key (respectively), and set the direction to backwards.
- Sequential retrieval-no key is supplied. Records can be returned in ascending or descending key order, commencing with the lowest key or highest key in the file respectively, and continuing forwards or backwards through the file until sequential retrieval is terminated.

Before a record can be retrieved (except for sequential retrieval) a key or partial key must be identified. This is done using the KEY= or KEYVAR= operand on the &FILE GET statement. It is possible to identify the key prior to the &FILE GET (for example, for sequential retrieval) by specifying the KEY= or KEYVAR= operand on a &FILE SET statement.

The success of the &FILE GET statement can be tested using the &FILERC system variable. If the &FILE GET statement was not successful the &VSAMFDBK system variable contains a standard VSAM completion code indicating the type of error that occurred. &FILERC signals end-of-file conditions as well as error conditions.

If generic retrieval or sequential retrieval is performed, the full key of the retrieved record is placed in the &FILEKEY system variable. For example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE SET KEY='REC'
.NEXTREC
   &FILE GET OPT=KEQ VARS=A*
   &IF &FILERC EQ 4 &THEN &ENDAFTER &WRITE DATA=END-OF-FILE
   &ELSE &IF &FILERC NE 0 &THEN &ENDAFTER &WRITE +
      DATA=ERROR=&VSAMFDBK
   &WRITE DATA=THE RECORD KEY RETURNED = &FILEKEY
&GOTO .NEXTREC
```

Continuation of a generic retrieval depends on the next &FILE statement. Any statement other than another &FILE GET, destroys the current file position, and ends the generic retrieval. If an &FILE GET statement is issued with a different key from that issued on the first generic &FILE GET, the generic retrieval ends, otherwise the key used on the statement is ignored, and retrieval is determined by the current file position.

When a generic read is performed (for example, when using the OPT=KEQ or OPT=KGE operand on &FILE GET), NCL maintains a generic environment until the NCL procedure specifically stops using generic retrieval, for example by issuing &FILE GET OPT=END. The maintenance of this generic environment holds the current position within the UDB, but in doing so necessarily uses VSAM buffer space. In procedures such as EASINET where there can be much concurrent UDB activity, it is important to keep the length of time that generic environments are open to a minimum and to avoid the use of generic processing where it is not necessary-for example, if &FILEKEY has been set to the full key of a record, do not use &FILE GET OPT=KEQ to read the record.

Restrictions When Using UDBs

When designing facilities that will use file processing the following must be taken into account:

- All records must be keyed, unless processing with an ESDS.
- Keys must range in size from 1 to 255 bytes.
- Keys for base clusters must start in position 0 (unless processing in unmapped mode) of the record. This is often termed relative key position (RKP) 0.
- Allowance should be made for field separators, or length and tag bytes added by NCL when determining record sizes for mapped or delimited files.
- The maximum record size is 32K when doing I/O from NCL variables. Otherwise it is the maximum record size for the file.
- When retrieving records, fields are returned in variables which have a maximum length of 256 characters. Creating fields in excess of this length might not be practicable, although a field in excess of 256 characters can span multiple variables after an &FILE GET.
- Unmapped format UDBs have no record structure maintained by NCL. The user is responsible for determining the record format of unmapped format files.
- Mapped and unmapped format UDBs can contain non-character data. The user must make allowance for this and might want to convert the data to expanded hexadecimal format after it is retrieved from an unmapped format UDB.
- If a UDB is of a format describable to Mapping Services using a map, then mapped format processing can be performed on it.

Create UDBs with Alternate Indexes

The description of UDB usage in the preceding sections has been restricted to the processing of records that are identified by a record key starting at position 0 in the record. This is referred to as the base key for the file.

The base key allows an NCL procedure to position a UDB to a particular record or group of records within the file, based on the comparison of a key provided as an argument against the keys of records present on the file.

This method of accessing records is adequate for many NCL applications. However, there are other applications which might require access to records within the UDB based on more than one argument.

The use of alternate indexes provides a means by which many arguments can be used to position a UDB to a particular record or record group and allows many different views of the same data held in a single UDB.

As an example, an installation that runs EASINET to provide access to all VTAM applications within its network uses the special LOGPROC NCL procedure to monitor the passing of EASINET controlled terminals to different applications. Each time a logon request is processed, a message is written to the activity log. It is intercepted by LOGPROC and a record written to a UDB that records the time, target application, and name of the terminal concerned.

This information is built up over time and NCL procedures are written to analyze the activity represented by the data on the UDB, where analysis is required by time, by application and by terminal name.

If the base key of the UDB starts with, for example, a time stamp, then the UDB is seen by NCL as being sorted in ascending time stamp order. In this case UDB processing using the time as an argument is very simple; however, if an analysis of the UDB by terminal name is required, the UDB can still be positioned only by the time argument, which is not what is required.

Alternate indices which allow the UDB to be viewed by NCL as being sorted in orders other than by time (for example, by terminal name or by target application name), simplify the processing required in procedures that need to analyze the UDB using arguments that are not satisfied by the base key of the file.

The following sections describe the steps required to create alternate indices for a UDB, the use of alternate indices within NCL procedures and the considerations associated with their use.

VSAM Considerations for Alternate Indexes

An alternate index is an individual VSAM data set, maintained by VSAM, which contains information and keys to the base cluster VSAM data set to which the index applies. It can be regarded as providing the user with a different view of the data within the base cluster. As changes are made to records on the base cluster, VSAM (provided that the correct options are defined) automatically updates the alternate indexes associated with the cluster.

VSAM relates an alternate index to its target base cluster with a definition known as a path. It is this path which NCL uses to retrieve data using keys from the alternate index.

A given base cluster can have many alternate indices, depending on the complexity of the record formats maintained on the UDB and the number of different ways those records are to be accessed.

The process to establish a new base cluster with an alternate index is:

- Define the base cluster using the IDCAMS DEFINE CLUSTER function.
- Load the base cluster. This is done using the VSAM IDCAMS utility REPRO statement.
- Define the alternate index using the IDCAMS DEFINE AIX function.
- Define the path using the IDCAMS DEFINE PATH function.
- Build the alternate index using the IDCAMS BLDINDEX function.

  This cannot be done for an empty base cluster.

  Add DD statements to the system JCL for both the base cluster and the path (but not the alternate index).
- Assign file IDs to UDBs using the UDBCTL statement.

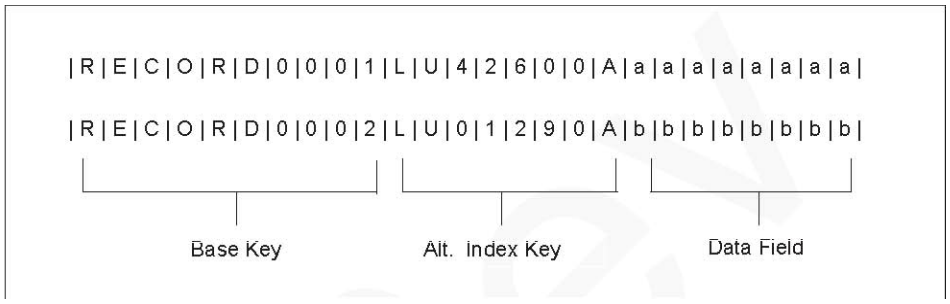
Notes:

- When defining a base cluster with an alternate index, both the base cluster and the alternate index must be defined with VSAM SHAREOPTIONS
- of (3 3). This is necessary as two ACBs will be used, one to reference the base cluster and a second to reference the path. A reference to a path causes both the base cluster and its associated alternate index to be opened. VSAM therefore sees two concurrent users of the base cluster.
- If duplicate keys are to be allowed on the alternate index, the NONUNIQUEKEY operand must be used when the index is defined. Duplicate keys on the alternate index are likely when the alternate index key is positioned over part of the base key of a cluster.
- To establish an alternate index for an existing base cluster, the same procedure is followed but with the omission of the first two steps (that is, defining and loading the base cluster).

Key Structures and Alternate Indexes

The record key used when processing a base cluster UDB can be 1 to 255 bytes in length and usually starts at offset 0 in the record.

When accessing a base cluster by an alternate index, a key is still used, but the key could be located anywhere within the record that it identifies. The precise location of the key within the record is defined (as an offset from the start of the record) when the alternate index is defined.


|R|E|C|O|R|D|0|0|0|1|L|U|4|2|6|0|0|A|a|a|a|a|a|a|a|a|
|R|E|C|O|R|D|0|0|0|2|L|U|0|1|2|9|0|A|b|b|b|b|b|b|b|b|
Base Key
Alt. Index Key
Data Field

This figure shows a representation of two records on a base cluster. The base key of each record starts at offset 0 and is 10 bytes long. In the example the base keys of the two records are:

```
RECORD0001
RECORD0002
```

An alternate index is associated with the base cluster and has been defined with a key that is 8 bytes long and starts at offset 10 in the record, that is, it follows the base key of the record.

In the example the alternate index keys have values of:

```
LU42600A
LU01290A
```

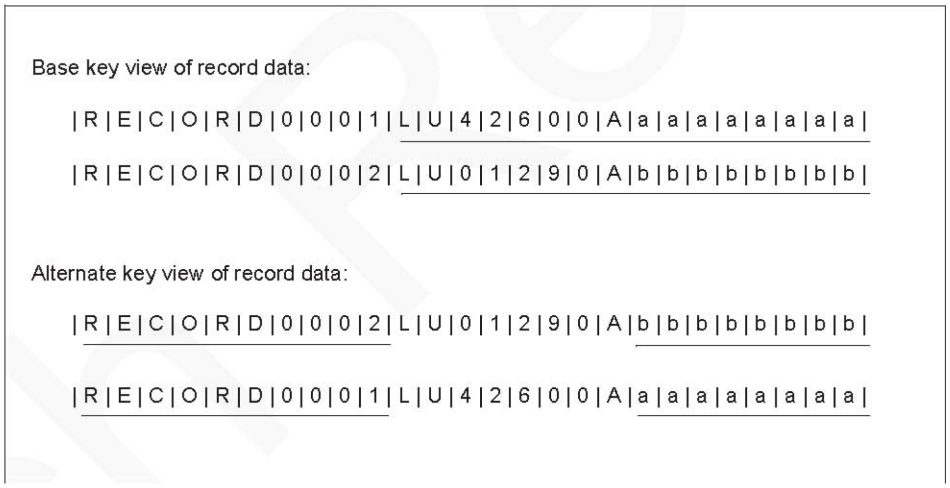
Additional alternate index keys can be defined which span different strings within the base cluster record.

Retrieve Data Using Alternate Indexes

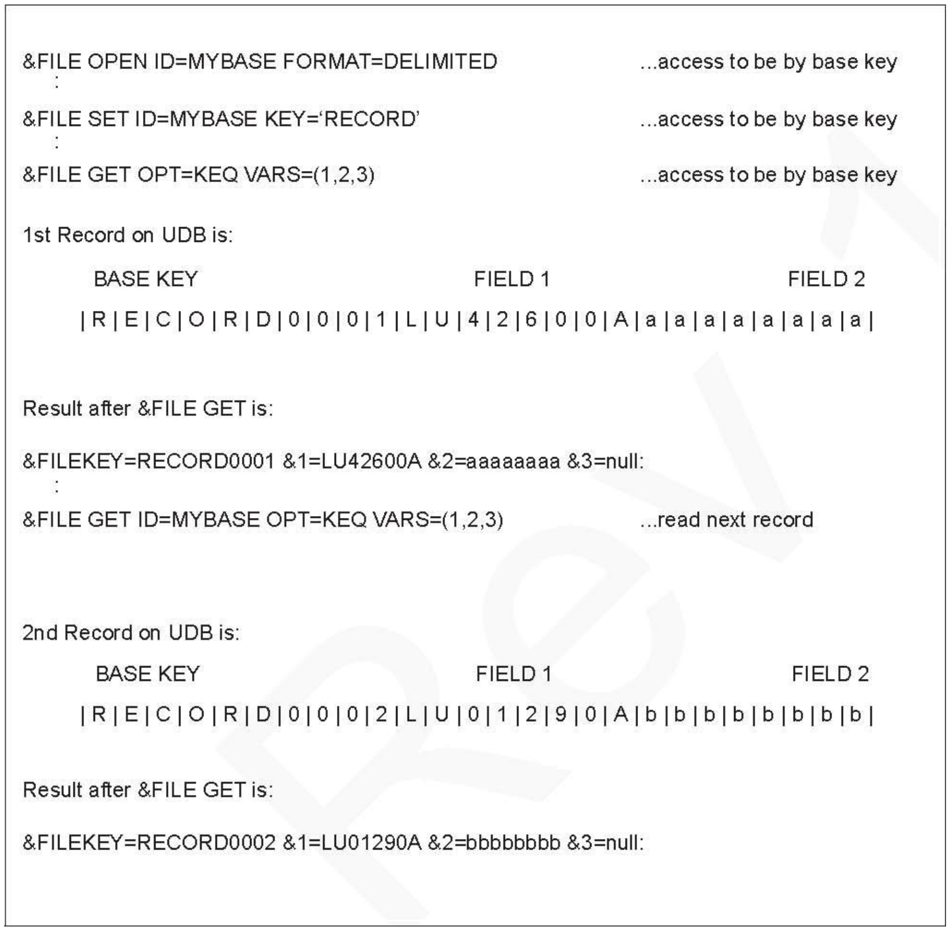
Access to UDBs via an alternate index is similar to standard base key access, but the way the data is presented to the NCL procedure after completion of the &FILE GET is different.

As with standard base cluster usage, the NCL procedure must nominate the UDB to be processed:

- For base cluster processing, the ID operand on the &FILE statement is used to identify the logical file ID associated with the DD statement that defines the base cluster data set in the execution JCL.
- For alternate index processing the ID operand on the &FILE statement identifies the logical file ID associated with the DD name that defines the Path to be used. (Remember that access to a cluster using an alternate index is achieved via a VSAM path definition only and that no direct processing is done on the alternate index cluster itself.)


Base key view of record data:
|R|E|C|O|R|D|0|0|0|1|L|U|4|2|6|0|0|A|a|a|a|a|a|a|a|a|
|R|E|C|O|R|D|0|0|0|2|L|U|0|1|2|9|0|A|b|b|b|b|b|b|b|b|
Alternate key view of record data:
|R|E|C|O|R|D|0|0|0|2|L|U|0|1|2|9|0|A|b|b|b|b|b|b|b|b|
|R|E|C|O|R|D|0|0|0|1|L|U|4|2|6|0|0|A|a|a|a|a|a|a|a|a|

This figure shows the different views of the data in two records on a base cluster that are seen when the UDB is accessed through the base key or through the alternate index key. (In each of the example records shown, the key portion is shown in plain text and the data portion is shown underlined.)



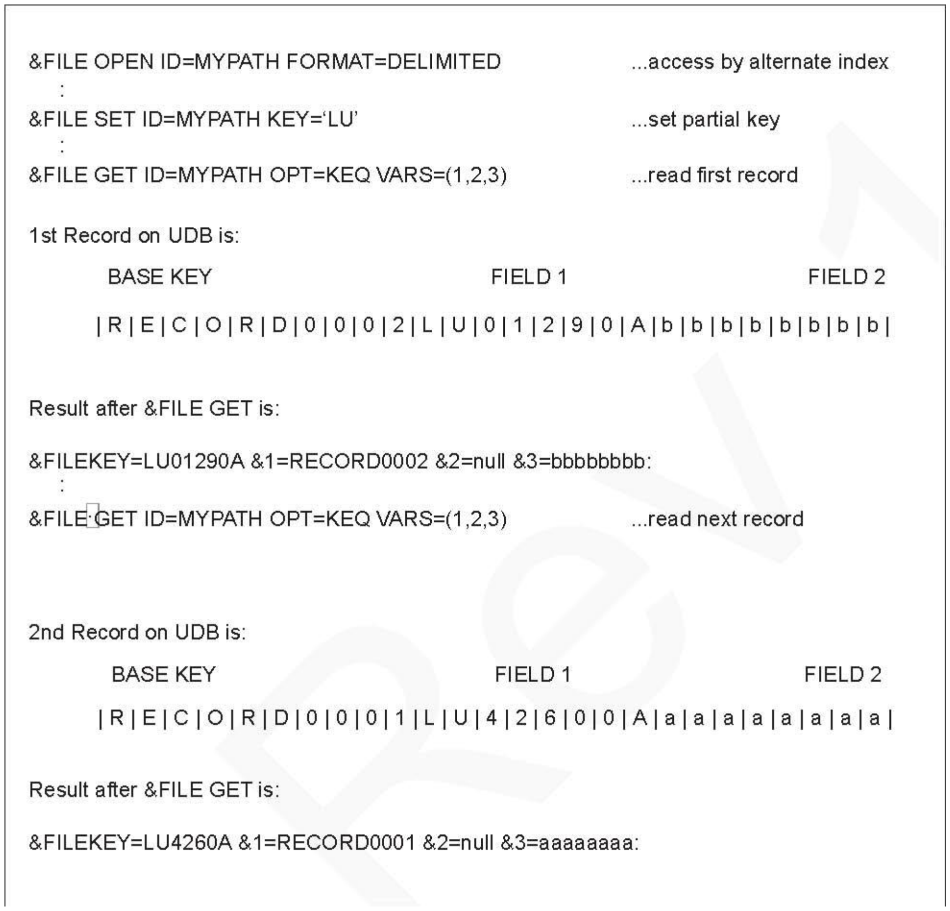
```
&FILE OPEN ID=MYBASE FORMAT=DELIMITED                   ...access to be by base key
     :
&FILE SET ID=MYBASE KEY='RECORD'                        ...access to be by base key
     :
&FILE GET OPT=KEQ VARS=(1,2,3)                          ...access to be by base key

1st Record on UDB is:

        BASE KEY                        FIELD 1                    FIELD 2

      | R | E | C | O | R | D | 0 | 0 | 0 | 1 | L | U | 4 | 2 | 6 | 0 | 0 | A | a | a | a | a | a | a | a | a |


Result after &FILE GET is:

&FILEKEY=RECORD0001 &1=LU42600A &2=aaaaaaaa &3=null:
     :
&FILE GET ID=MYBASE OPT=KEQ VARS=(1,2,3)                ...read next record


2nd Record on UDB is:

        BASE KEY                        FIELD 1                    FIELD 2

      | R | E | C | O | R | D | 0 | 0 | 0 | 2 | L | U | 0 | 1 | 2 | 9 | 0 | A | b | b | b | b | b | b | b | b |

Result after &FILE GET is:

&FILEKEY=RECORD0002 &1=LU01290A &2=bbbbbbbb &3=null:
```

Rev

This figure show an example of reading records by using the base key. In the example shown in these figures, the UDB is a UDB (or delimited) format file and each record contains two data fields. The data fields are each terminated by a field separator (X'FF'), although these are not shown in the diagrams. The alternate index key in this example is chosen to overlay exactly the first data field, but does not include the field separator character at the end of the field.

The base key's view of a record on the UDB is that the first 10 bytes of a record are its key and that everything else is data. The alternate index view shows that everything is data, apart from the 8 bytes starting at offset 10 in each record. following example shows the effect of these conflicting views on the data actually presented to an NCL procedure on completion of an &FILE GET statement.

For the example, we will assume that the logical File ID to be used when accessing the base cluster directly is MYBASE and the logical file ID used to access the base cluster via its alternate index is MYPATH.



```
&FILE OPEN ID=MYPATH FORMAT=DELIMITED            ...access by alternate index
    :
&FILE SET ID=MYPATH KEY='LU'                      ...set partial key
    :
&FILE GET ID=MYPATH OPT=KEQ VARS=(1,2,3)          ...read first record

1st Record on UDB is:
          BASE KEY                        FIELD 1                 FIELD 2

      | R | E | C | O | R | D | 0 | 0 | 0 | 2 | L | U | 0 | 1 | 2 | 9 | 0 | A | b | b | b | b | b | b | b | b |


Result after &FILE GET is:

&FILEKEY=LU01290A &1=RECORD0002 &2=null &3=bbbbbbbb:
    :
&FILE GET ID=MYPATH OPT=KEQ VARS=(1,2,3)          ...read next record



2nd Record on UDB is:
          BASE KEY                        FIELD 1                 FIELD 2
      | R | E | C | O | R | D | 0 | 0 | 0 | 1 | L | U | 4 | 2 | 6 | 0 | 0 | A | a | a | a | a | a | a | a | a |


Result after &FILE GET is:

&FILEKEY=LU4260A &1=RECORD0001 &2=null &3=aaaaaaaa:
```

The previous figure shows how data is presented to an NCL procedure that reads records from the example UDB, using the base key to access the file. As noted before, it is assumed here that the alternate key is in fact a separate field within the record, although this does not have to be the case.

This figure shows the same sequence of events, but with access to the UDB being via the alternate index. There are two significant differences in the results of the &FILE GET statements compared with using the base key:

- The order of the records is different.
- The data presented in the &FILE GET variables is different.

Records on the UDB are always arranged in ascending key order, according to the key being used to access the UDB.

Therefore, when using the base key the order of the records on this example UDB is:

```
RECORD0001........
RECORD0002........  and so on.
```

However, when viewed from the perspective of the alternate index, the order of the records on the UDB has nothing to do with the value of the base key; as far as the alternate index is concerned the UDB is arranged in ascending order of the alternate keys of the various records on the file. As a result, when accessing records via the alternate index, the order becomes:

```
LU01290A.......
LU42600A.......  and so on.
```

and so, as shown in the figures, the order in which the physical records are retrieved from the UDB depends upon which key is being used to access the file.

Control UDB Availability

As described earlier, UDBs are made available to NCL procedures through the UDBCTL command, which assigns a logical file ID. For more information about the UDBCTL command, see the Online Help.

For a variety of reasons you might want to change the availability of UDBs from time to time. The UDBCTL command provides the following facilities:

Stop a UDB

This logically blocks all further attempts to access the specified UDB from any procedure, so that no further &FILE statements which specify the ID for this UDB will be allowed. The physical data set remains open. STOP is rejected if there is any current user of the file.

Close a UDB

Implies a STOP and physically closes the file. This might be necessary if external updating of the UDB is required while the file is still allocated to your product.

Reset a closed UDB

In VSAM terms, RESET causes your product to open the VSAM ACB with the RST option, which has the effect of emptying the entire file. VSAM constraints mean that RESET cannot be used on UDBs unless they are sub allocated (in those levels of VSAM that support sub allocation) and is not allowed if the cluster is defined with KEYRANGES or has alternate indexes associated with it.

Open a UDB

Reopens a previously-closed UDB. If the UDB had also been RESET and has not been loaded externally before re-opening, Your product will initialize (see page 170) the file. A UDB that is OPENED requires re-assignment of its logical file ID before it becomes available for processing again.

These UDBCTL options provide operational control over the availability of UDBs to the product region. UDBCTL can itself be issued from within NCL procedures by suitably-authorized users.

Work with Files

NCL includes an &FILE verb that provides the high-level interface to its file processing facilities. The verb has seven major operands:

&FILE OPEN

Opens a file and identifies its processing mode

&FILE SET

Sets the default file ID, sets the full or partial key, sets the processing mode for subsequent processing

&FILE ADD

Adds a record

&FILE GET

Retrieves a record

&FILE PUT

Adds or updates a record

&FILE DEL

Deletes a current record

&FILE CLOSE

Releases file processing connections

The following system variables are provided to help test the success of functions:

&FILERC

the completion code for the last function

&FILERCNT

the number of records processed by the last generic delete (&FILE DEL) operation

&VSAMFDBK

the VSAM RPL feedback code from the last function

&FILEKEY

the key of the last record referenced

Note: For more information about the &FILE verb and system variables, see the *Network Control Language Reference Guide*. For more information about the UDBCTL command, which is used to control the availability of UDBs for processing, see the online help.

Logical File Identifiers

When using NCL, files are referenced by a logical file identifier or file ID. The file ID provides a logical connection to the physical data set. A file ID must be assigned using the UDBCTL command before a file can be referenced by an NCL procedure.

Suppose, for example, that you want to process a file called HELPDESK.DATA, which is defined by using the ALLOCATE command with a DD name of HELPDB1 and all NCL procedure references to the physical file are to be made to the symbolic name HELPDESK.

Before the file HELPDESK.DATA can be processed by NCL, a UDBCTL command must be executed specifying the symbolic name by which this file will be referenced from NCL procedures. This UDBCTL command relates the DD name of the data set with its symbolic name and is coded:

```
UDBCTL OPEN=HELPDB1 ID=HELPDESK options
```

For a particular NCL procedure to access the file that has now been assigned the symbolic, or logical name of HELPDESK, the procedure must contain an &FILE OPEN statement specifying the file that is to be processed:

```
&FILE OPEN ID=HELPDESK FORMAT=DELIMITED
```

This approach offers considerable flexibility when it becomes necessary to change the system configuration, since all NCL procedures reference files using their logical file ID and are therefore shielded from external changes.

Should a data set become full, a single UDBCTL command can simultaneously migrate all NCL procedures to reference a new data set without change to the NCL procedures. Consider our example using the HELPDESK file. It might be desirable at the end of the day to swap to another data set and perform other processing on the previous day's data while continuing to record new problems in another file. You would allocate multiple data sets and use a UDBCTL command at the appropriate time to swap to an alternative data set. For example:

```
UDBCTL STOP=HELPUDB1(stop current use)
UDBCTL OPEN=HELPUDB2 ID=HELPDESK(swap to new file)
```

The NCL procedures continue to use the same ID on the &FILE statement, but now use a different physical data set as the UDBCTL command has changed the logical relationship:

```
&FILE OPEN ID=HELPDESK FORMAT=DELIMITED
```

```
(NCL procedure is unchanged)
```

Release File Processing Resources

The &FILE OPEN statement allocates certain resources to the requesting NCL procedure. It is not normally necessary to release file processing resources within an NCL procedure. This is performed automatically when the NCL procedure terminates.

Under certain circumstances, such as in an EASINET procedure, where there might be many concurrent users performing file processing, it might be desirable to release any file processing overheads when they are no longer required, to ensure that system overheads are minimized.

This can be accomplished using the &FILE CLOSE statement. &FILE CLOSE allows specific files or all files to be freed. When this is done, any storage associated with processing the file is released and the connection is logically severed for that user.

Having used &FILE CLOSE to release a particular file, connection can be re-established using another &FILE OPEN statement.

&FILE CLOSE destroys any generic retrieval position a user might have established within a file and any subsequent reference would have to reestablish that position if required.

Display File Information

The SHOW UDB command can be used to display details about files available to the system. This information includes details about the number of active users, space usage, and the status. In addition, any open error codes that caused a file to be disabled are displayed.

The SHOW VSAM command is used to display the VSAM attributes of the various VSAM data sets in use by your product. Information presented by this display includes:

- Record sizes
- Control Interval sizes
- Control Interval and Control Area split statistics
- Current index and data buffer allocations
- Information about string and buffer shortages
- LSR pool statistics (in z/OS systems)

The SHOW UDBUSER command is used to obtain information about current usage of particular UDBs in the system. The display lists all UDBs showing DD names, file IDs, and the names of the user IDs and NCL procedures that are currently accessing each UDB.

Note: For more information about the SHOW UDB, SHOW VSAM, and SHOW UDBUSER commands, see the online help.

Specify the File Processing Mode

The &FILE statement has a mandatory ID=*file_id* operand. This operand identifies the UDB on which the NCL verb acts. The ID operand enables several files to be open and processed simultaneously. You specify a FORMAT operand with the OPEN or SET operands (that is, &FILE OPEN ID=... FORMAT=... and &FILE SET ID=... FORMAT=...). The FORMAT operand is used to specify the current processing mode for a file (delimited, unmapped, or mapped). You can switch between two file formats without losing position.

Specify the File Key

You can specify a file key on an &FILE statement in several ways. The way in which the value is specified on the KEY= operand is independent of the current processing mode (delimited, mapped, or unmapped). The value of the KEY= operand is interpreted as a character string by default. The value can be specified inside quotes, and substitution is carried out on any variables contained inside the quotes. You can put the letter X after the quotes to indicate that the data between the quotes is to be hexadecimal-packed to create the key.

Example: Specify the File Key

If &A=AAAA, both examples indicate a key of 'AAAA 0001':

```
KEY='&A 0001'
```

```
KEY='C1C1C1C140F0F0F0F1'X
```

also indicates a key of 'AAAA 0001'

There is also a KEYVAR= operand on the &FILE verb, which is an alternative to the KEY= operand. The KEYVAR operand is used to specify a variable whose contents is used as the key. For example, KEYVAR=A indicates that the contents of the variable &A is used as the key. In this case, if &A contains non-printable characters or trailing blanks, they appear in the key unchanged.

Work with Data

As with other NCL functions, data manipulated by file processing is usually maintained in tokenized form (for example, as NCL variables). The &FILE verb also enables data to be maintained in an MDO, but only for mapped format file processing. A typical NCL procedure might accept input from a full-screen panel in a series of tokens or variables and then add this data to a UDB. Alternately, it can receive data from another application using APPC into an MDO, and then add this data to a mapped format UDB.

The physical representation of the data on the UDB depends on its format:

- For mapped format files using $NCL map, the record consists of a list of vectors. Each vector has a two byte length field followed by a two byte tag which is followed by the data from the token. The entire list is encapsulated with its own two byte length and tag fields. For more information about the data structure and tag values, see the distributed $NCL map.
- For unmapped format files, the file processing statements still use variables, but NCL does not regard each variable as a field within the record. When writing a record to an unmapped format UDB all variables are written contiguously (that is, no field separators are inserted). The structure of the data within the record is therefore the responsibility of the user.

- For delimited format files, each variable is treated by NCL as a field within a record, each field being separated from its neighbor by a field separator. When data is retrieved it is returned in a series of variables using the field separators to determine the size of each field. This approach relieves the procedure of having to define the format and structure of data in UDBs.

When data is retrieved from an unmapped format UDB, NCL again makes no attempt to identify individual fields within the record. The entire record is read from the UDB and then split into as many variables as are necessary to hold the data that has been read. Alternatively, the user can indicate how many bytes of data from the record are to be placed into each variable on the &FILE GET statement.

When data is retrieved from a mapped format UDB it is placed into an MDO unchanged from the way it existed on the file. It is also possible to specify the name of a map on the file verb which will be attached to MDO when data is placed into it. When writing to a file using mapped format, the contents of the MDO are simply placed on the file as they are.

Data Set Positioning and Generic Retrieval

NCL supports both sequential and generic retrieval from keyed data sets. Such functions imply that a current position within the file is maintained so an NCL procedure can simply request the next record and it will be supplied. No incrementing of keys by the NCL procedure is necessary.

Under certain circumstances, such as with generic retrieval, it might be necessary to alter the retrieval sequence and commence retrieval using a different key.

NCL must be informed that such a change is required and that the current retrieval sequence is to be stopped. This is done using the &FILE GET ID=name OPT=END statement. This indicates to NCL that generic retrieval is to be terminated in anticipation of some other processing.

If an end-of-file condition is signaled, no &FILE GET ID=name OPT=END is required. The use of a non-generic function, such as the specific retrieval of a record, or the use of the KEY= or KEYVAR= operand on the &FILE verb to set the current key, will also cancel a previous generic function.

You can use KEY=' ' and the generic option together, in which case the key is used to gain initial position for the generic retrieval. Subsequent generic retrieval requests continue from this position, as long as the key supplied (if any) is the same as the initial key.

Mapped Format Files: Data Representation

When a file is processed using the mapped format processing option, a map is used to describe the arrangement of the records. The map describes the records in terms of structures or components. Maps are managed by Mapping Services and exist as separate entities within your product. When carrying out a retrieval using mapped format processing, the contents of the file record are usually placed into an MDO, and a map is then used to interpret the contents of the MDO.

It is also possible to read and write NCL tokens using mapped format processing. This is a special case of mapped format processing and a special system map called $NCL is used for this purpose. When NCL variables are written to a file using mapped format processing, and the $NCL map, they appear on the file as sub vectors. The sub vectors consist of a header followed by data. There is one sub vector for each token written to the file. The sub vector headers consist of a 2 byte length field followed by a 2 byte key where the key is X'0000'. (For example, a variable containing the value X'C1C1' would appear on a file record as: X'00060000C1C1', where 0006 is the length, 0000 is the key, and C1C1 is the data.)

The process of writing variables to a mapped format file using the $NCL map is reversible. This means that if a set of variables is written to a record under these conditions, and the contents of this record are then read back into the variables at some other stage, the original values contained in the variables will be the restored.

Note: This allows tokens containing non-printable characters to be handled, that is, data transparent.

Unmapped Format Files: Data Representation

Unmapped format UDBs, which are usually files created or processed by mechanisms other than NCL, can contain noncharacter hexadecimal data (for example, records can contain strings of binary zeros). When data is read from a file using unmapped mode, the entire record is read in byte-for-byte as is, and non-printables remain as they appeared on the file. The data is placed into variables in lots of 256 bytes (the maximum variable size) unless otherwise specified. If the data contains non-printables, expand the data to hexadecimal using the &HEXEXP statement before processing it. You can specify a length of 128 for each variable on the &FILE GET statement, so that no overflow occurs when the data is hexadecimal expanded.

Note: Most data formats are most conveniently processed using mapped format files and maps developed using Mapping Services.

When writing to an unmapped file, the contents of the output buffer are placed onto the file unchanged, including non-printables. If VARS= or ARGS is specified on the &FILE PUT/ADD statement, the contents of each of the variables are concatenated and placed onto the file. Any non-printables that were contained in the variables are also placed onto the file. If DATA= is specified on the &FILE PUT/ADD, substitution takes place on the buffer before it is written to the file.

If any of the variables specified following the DATA= operand contain non-printables, these are preserved. Spaces and text between variables is also written to the file.

For example, if the variable &A contains X'C100C1' and the variable &B contains X'C200C2', then after the following statement is executed:

```
&FILE PUT ID=MYFILE DATA=&A XXX &B
```

the following result appears on the file:

```
X'C100C140E7E7E740C200C2'
```

One of the principal uses for unmapped format processing is in the dynamic preparation of reports from NCL procedures (particularly on z/OS systems):

1. An ESDS UDB is allocated dynamically to SYSOUT.
2. Report lines are written in unmapped format.
3. The UDB is then closed for immediate printing.

When using &FILE PUT/ADD to write to SYSOUT file, carriage control options are available to assist with report formatting.

Note: For more information about the PUT and ADD operands, see the &FILE verb in the *Network Control Language Reference Guide*.

DBCS Considerations When Using Files

If you write data to a UDB that represents a DBCS data stream, the data stream is written to the file using shift characters that depend on the DBCS mode in which your product is operating. This is determined by the SYSPARMS DBCS= operand. A system that reads that file must therefore operate in the same DBCS mode as the system that originally wrote the data, otherwise the reading system will not be able to recognize the shift characters present in the data.

&FILE GET Statement and Unmapped Format UDBs

The &FILE GET statement is used to retrieve data from any type of UDB and when issued causes NCL to present data to the NCL procedure in a series of variables, or as an MDO.

It is only possible to use the MDO= operand for mapped format processing. As described earlier, if the UDB is a delimited format file, each variable on the &FILE GET statement is returned with the contents of field, where the fields are segments in the record delimited by X'FF'.

However, if the UDB is an unmapped format UDB, NCL cannot provide the data to the procedure as a set of individual fields-NCL has no knowledge of field boundaries within the record.

Therefore, for unmapped format UDBs, NCL treats the record read from a UDB as a single string and then splits this string into as many full-length variables as are required to hold all the data, or as many as are provided on the &FILE GET statement, whichever is the smaller number.

A full-length variable is 256 characters long. It is also possible to indicate explicitly how many bytes are to be placed into each variable.

For example:

```
&FILE GET ID=MYFILE VARS=(A(10),B(100),C(40))
```

indicates that the first 10 contiguous bytes of the record being read will be placed unchanged into the variable &A, the next 100 bytes will be placed into variable &B, the next 40 bytes will be placed into variable &C, and any remaining bytes will not be placed into any variable (that is, they will be ignored).

The following shows an example of using &FILE GET on an unmapped format file.

```
&FILE OPEN ID=MYFILE FORMAT=UNMAPPED
&FILE SET ID=MYFILE KEY= 'RECORD1
'&FILE GET ID=MYFILE ARGS

Record contents
|C1|C2|C3|15|C4|15|F3|C5|15|15|15|F1|15||C1|C2|C3|15|C4|15|F3|C5|15|15|15|F1|15|

:&1 contents after &FILE GET:
|C1|C2|C3|15|C4|15|F3|C5|15|15|15|F1|15||C1|C2|C3|15|C4|15|F3|C5|15|15|15|F1|15|
```

Data Conversion and Unmapped Format UDBs

If data on an unmapped format UDB is expected to contain non-printable characters it is wise after reading the data into variables to convert it to an expanded hexadecimal form to preserve the non-printables (this is because many NCL verbs automatically translate non-printables to blanks-X'40's).

Regardless of the data representation format selected on the &FILE OPEN/SET statement, data read from or written to an unmapped format UDB might require conversion from expanded hexadecimal format to character format, or conversion from character to expanded hexadecimal format. Two NCL statements, &HEXPACK and &HEXEXP provide this facility.

For example, where &HEXEXP is being used for file processing and data is being read from a file which has records which consist of a 1 byte error code followed by text. If the error code is not X'00', then there is a problem with the record.

A record could be read from the file as follows:

```
&FILE OPEN ID=MSGFILE FORMAT=UNMAPPED
&FILE GET ID=MSGFILE KEY='00000001'X VARS=(ERRCODE(1),MSGTEXT)
&TEMP = &HEXEXP &ERRCODE
&IF &TEMP GT 00 &THEN &GOTO .ERROR
   .
   .
   .
&WRITE DATA=FOLLOWING MESSAGE READ FROM FILE: &MSGTEXT
```

The record with a key of X'00000001' was read into two variables, &ERRCODE and &MSGTEXT. A subscript was used to indicate that the first byte was to go into &ERRCODE, and the remaining bytes (up to 256) were to be placed into the variable &MSGTEXT.

An &HEXEXP statement was used to convert the &ERRCODE variable from binary (Packed hexadecimal) format into NCL format. (For example, if &ERRCODE was X'00', &HEXEXP would convert it to X'F0F0'). This enabled the value of it to be tested for an error condition.

Note: A map could be used to describe the file, and then the record could be read into an MDO using mapped format.

Key and Data Differentiation

When a record is read from the UDB, whether it is a delimited format or an unmapped format file, NCL normally performs the following steps when preparing the data for presentation to the procedure, regardless of the key under which the record has been retrieved.

- The record is read as a single string.
- The full key of the record is extracted and placed in &FILEKEY.
- If the file being processed is a delimited format file and the record is not read under the base key, the section of the record previously occupied by the key is replaced by a field separator (X'FF').
- The remaining data is placed into the variables nominated on the &FILE GET statement according to the rules applicable to the type of UDB being processed.

The significant point here is that the key under which the record is retrieved is not, by default, regarded by NCL as part of the record data and does not therefore appear in the &FILE GET variables.

The result of this is that the same fields within a given record can be relocated into different &FILE GET variables, depending upon which key is used to read the record.

Previous figures show that when the records in the example are read by successive &FILE GET statements, the two separate fields of which the records are composed are placed into the two variables &1 and &2, as expected. &3, specified on the &FILE GET statement, is set to null since no data was available to place into it.

However, a different &FILE GET results when the alternate index is used to read the same two records. NCL has obeyed the rules for data presentation with the following result:

- The record is read under the alternate index key.
- The key used (LU01290A) is extracted and placed in &FILEKEY and replaced with a X'FF' field separator. Remember that the key in this case is followed by a field separator too, so now there are two consecutive field separators in the record.

- The data is placed into the nominated variables field-by-field, starting at the left-hand end of the record. This results in values:
  - &1 = RECORD0002
  - &2 = null
  - &3 = bbbbbbbb
  - The reason that &2 is null is because that which used to be the key value has been extracted from the record and replaced with a field separator. As NCL scans the data to determine the location of the various fields, two consecutive field separators are found indicating the presence of a null, or zero-length field. Consequently, &2 becomes a null variable. The result of this is that the data (bbbbbbbb) is no longer located in the same variable as it was when the record was read under the base key.

Note: When processing with alternate indices the positioning of the alternate key can significantly impact the way in which an NCL procedure must process the UDB. It is therefore recommended that you familiarize yourself with this process and try a few simple examples first, to ensure you understand it.

Key Extraction Options

The rules for data presentation allow accurate prediction of what each &FILE GET variable contains regardless of which key is used to retrieve a record. However, the effects of key extraction from record data can sometimes prevent the use of common processing logic when the same procedure uses different keys to access the same UDB.

The &CONTROL option NOKEYXTR lets you leave the key in the data portion of the record (as well as placing a copy of the key in &FILEKEY as usual). This option applies only when reading data using a key that does not start at offset 0 within the record.

The effect of NOKEYXTR is to eliminate null variables occurring after an &FILE GET (where the null represents the original location of the key). NOKEYXTR also eliminates the splitting of single fields into two when the key happens to overlay part of a single field.

Use NOKEYXTR in procedures that reference a UDB under multiple keys where you want to provide common processing routines for specific fields within the records.

Update Restrictions on Alternate Indexes

NCL supports the retrieval of data across any number of alternate indexes. However, data updating is restricted to the base cluster for delimited format files. If a record that has been retrieved using an alternate index is to be updated, the ID operand on the &FILE statement is used to indicate that processing has swapped to the base cluster, and that the necessary key must be set (either on an &FILE SET or on the &FILE PUT) before the record can be written back to the UDB.

Even when using the base cluster, the base key must start at offset 0 within the record (RKP 0) for a UDB (or DELIMITED) format file. No updating is allowed for any cluster which violates this rule.

This restriction does not apply to unmapped format UDBs which can be updated even if the key does not start at offset 0. In this case if insufficient data is supplied to the left of the start of the key, null padding (X'00') is added up to the start of the key location.

Offline Processing of Data Sets

On z/OS systems, the DEALLOCATE command can be used to release a UDB for offline processing. Before you deallocate the UDB, it must be closed using the UDBCTL command.

This can only be performed once all current users have ceased using the UDB. If deallocation cannot be used, the stripping of files created by NCL for further offline processing can be achieved without a restart of the system if the following rules are followed:

- VSAM SHAREOPTIONS must allow concurrent access to the data set.
- DISP=SHR must be specified on the data set (Z/OS only).
- The UDBCTL CLOSE=xxxxx operand is used to stop further logical connections to the file and to close the physical data set. This is allowed only if there are no users currently referencing the file.
- Use a utility to strip the file (for example, the VSAM REPRO utility). Any offline processing programs must take any high-value (X'FF') field separators into account when determining record formats on UDB format files.

Back Up Online Data Sets

While standard batch utilities can be used to back up offline VSAM data sets, they cannot be safely used to back up active online data sets.

Use an offline utility to back up only data sets that are closed and preferably deallocated from your product.

If an offline utility is used to back up an active online data set, the backup completes as though it were successful. A later restoration of that data set could result in a corrupted file. The offline utility has no knowledge of the state of the file, or any in-storage indexes or buffers that could have been written to the file at the time of the backup.

To resolve this problem, perform online backups of active data sets. You can use the Dataset Services UTILITY option to invoke the IDCAMS utility and perform a backup. When operating in this manner, the online utility has access to the in storage buffers and indexes, and can guarantee a usable backup of the data set.

Note: For more information about Dataset Services options, see the *Network Control Language Reference Guide*.

For operational reasons, it is still desirable to halt activity on a file temporarily so that a precise recovery time is known.

Chapter 9: System Level Procedures

This section contains the following topics:

System Level Procedures

NCL can be used to write private programs that are executed by command from your OCS windows, or implicitly from options such as the User Services panel.

Another class of NCL process called system level procedures executes exclusively in the NCL processing regions of certain background virtual users. These system level procedures have access to the special information flows that occur within your product region, namely:

- The activity log
- VTAM messages
- EASINET terminal control data
- OCS window traffic

Activity Log

The stream of output messages to the activity log provides a serial record of all activity in the system. Therefore, it is an ideal source of real-time information about system events. While the log is accessible for on-line review by real operators, there is a special background virtual user environment called LOGPROC that directly accesses every message written to the activity log through special NCL verbs:

- The LOGFILES Customizer parameter group nominates the name of a procedure that is to act as the LOGPROC procedure. As such, it can execute these special verbs to see and modify messages destined for the activity log.
- LOGPROC is a system level procedure (one per system) that can act as a central intelligence point to monitor events being reported to the log. As such, it can observe and react to unusual or critical conditions, and provide a platform for automatic recovery action.

VTAM Messages

The VTAM primary program operator (PPO) interface lets your product receive unsolicited messages from VTAM about network events that occur for reasons other than from operator commands. For example, if an NCP fails, VTAM will generate unsolicited messages notifying the PPO application of the occurrence.

The PPO interface can also be fed with commands and responses resulting from operator action at the system console and from OCS windows within the region.

The stream of PPO messages represents a serial record of all unexpected events within the VTAM network. As such it is an ideal source of information about real time network failures.

While PPO messages can be routed automatically to OCS and system console operators for their attention, there is also a virtual user environment within your product called PPOPROC that directly accesses every message (or a chosen subset of messages) received from VTAM. This PPO message flow is accessed only through special NCL verbs:

- The SYSPARMS PPOPROC command nominates the name of a procedure that is to act as the PPOPROC procedure, and as such can execute these special verbs to see and modify PPO messages received from VTAM.
- PPOPROC is a system level procedure (one per system) that acts as a central intelligence point to monitor events being reported by VTAM. As such, it can observe and react to unusual or critical conditions and provide a platform for the collection of additional information about the event and for the initiation of automatic recovery action.

EASINET Terminal Control

The EASINET feature lets you place all idle network terminals under the control of your product whenever they are not natively logged on to another network application.

The logic of the EASINET procedure determines how the feature handles terminals.

Note: For more information, see the *CA SOLVE:Access Session Management Administration Guide*.

OCS Window Traffic Handling

Each user operating one or two OCS windows is entitled (depending on their user ID profile) to nominate a system level procedure to help them filter and monitor messages sent to those windows.

These procedures are called MSGPROCs. The name of the procedure executed for a user to act as the MSGPROC for each user OCS window is defined as part of their user profile:

- A MSGPROC procedure has access to the serial stream of messages sent to the OCS window for which it is executing and can receive, modify, delete, or react to any message or message group sent to the window.
- MSGPROC procedures can therefore act as preprocessors for messages that an operator would otherwise have to monitor.

Message Profile Concept

System level procedures access the messages flowing on their particular stream by issuing specialized NCL verb read requests that are unique to each type of system level procedure. For example, PPOPROC uses &PPOREAD to obtain the next PPO message.

The processing that each system level procedure undertakes depends on analyzing the messages that are supplied to it. To help analyze the message received and decide the specific processing required, the concept of message profiling is used.

A message that satisfies a system level procedure read request has a profile containing the attributes associated with that message. For example, a MSGPROC message profile includes not just message text but other information about color, highlight options, or message origins that the MSGPROC procedure might want to know or change or base some decision upon.

On completion of the read statement, all applicable message attributes are made available to the system level procedure as a series of reserved system variables known as message profile variables. An examination of the message profile variables is often sufficient to make a decision on whether to perform further analysis of the message, or to show that the message is of no interest to the procedure.

More information:

Message Profile (see page 472)

Intercept Solicited and Unsolicited VTAM Messages (PPOPROC Procedures)

The primary program operator interface (PPO) in VTAM can be defined so that your product receives both solicited and unsolicited VTAM messages. An NCL interface, named PPOPROC, is available to the VTAM PPO facility. PPOPROC can use this information to monitor status changes of VTAM resources. By being aware of status changes, PPOPROC can be the primary source of decisions controlling automated reaction to events that occur within the network.

PPOPROC has four possible sources of information:

- The VTAM PPO interface that delivers both unsolicited VTAM-generated messages and optional, solicited messages from VTAM commands issued at system consoles

  This source depends on the VTAM PPOLOG initialization parameter.
- Copies of commands and command responses resulting from operator activity
- Messages delivered to PPOPROC from &PPOALERT statements issued by another NCL process

  This facility generates test messages, and delivers them to PPOPROC for development and testing. The facility also lets you generate a new PPO message with more or less information than the original.
- PPOPROC-related messages of the three preceding types routed from remote regions across the Inter-System Routing (ISR) facility

PPOPROC messages from these sources can be used to report network resource status changes to a central point. The change can be unexpected (in which case VTAM reports the event as an unsolicited message) or in response to an operator command.

Centralization can be achieved by routing all relevant PPO messages to a central region. Several ISR options are used to facilitate this centralization.

You set up the central region for monitoring PPO messages with the ISR ENABLE=PPO UNSOLICIT=INBOUND command. This setup allows the region to receive PPO messages from remote regions. The remote regions specify ISR ENABLE=PPO UNSOL=OUTBOUND to enable delivery of messages to the central PPO processing link.

Filter Messages Seen by PPOPROC

PPOPROC is not necessarily interested in every message that could be sent to it. To limit the number of messages PPOPROC has to process, your product uses the message definition table (DEFMSG table), which specifies VTAM message numbers that PPOPROC wants to see.

Very high message rates can occur at the PPO interface. Therefore the more you can decrease the number of messages delivered by the DEFMSG filter, the more efficiently your system will perform.

Modify the Message Definition Table: DEFMSG Command

The DEFMSG command lets you control message delivery for solicited and unsolicited VTAM messages. Three delivery destinations can be specified for each message:

- PPOPROC
- LOCAL (that is, to OCS operators)
- REMOTE (that is, systems linked by ISR)

In addition, EDS events can be defined in the DEFMSG table for VTAM messages.

A default class of messages has been pre-defined to the message table and initially set for delivery to all destinations. The delivery options for the default class messages can be altered using the DEFMSG command; however they always remain in the default class.

Message Filtering: Solicited Messages

Solicited messages are those generated as a result of VTAM commands. VTAM commands can be sourced from a system console or issued by OCS operators or NCL procedures. These messages always pass through message definition table (DEFMSG) processing. Only those messages indicating PPOPROC delivery in the DEFMSG table will be passed to PPOPROC for processing.

Note: Because they are normally displayed to the operator making a specific request, these messages are never delivered to other LOCAL receivers.

Message Filtering: Unsolicited Messages

Unsolicited messages are generated by VTAM to notify of an unexpected status change in the network-that is, a change which has not resulted from operator action.

Unsolicited messages are only delivered to the destinations specified for the VTAM message numbers in the DEFMSG table. If no delivery options have been set for an unsolicited message number, the delivery options for the unsolicited class entry are used, which, by default, are not set. To specify the delivery options, issue a DEFMSG UNSOL DELIVER=delivery command.

Design a PPOPROC Procedure

A PPOPROC procedure should be written as a closed loop. When invoked, it processes until the first &PPOREAD statement is detected and then suspends processing until a message arrives.

As each message arrives, the procedure logic should branch to an appropriate processing point for that message. After processing the message or a group of associated messages, the procedure should then return to the initial &PPOREAD where processing is suspended once again.

PPOPROC should be designed to START worker processes to analyze any serious events reported and initiate recovery procedures so the main PPOPROC process remains free to continue monitoring the unsolicited VTAM message flow.

Important! Be particularly careful when using verbs that are dependent on events outside PPOPROC control (such as &INTCMD followed by &INTREAD) as these can tie up PPOPROC processing and result in long PPO message queues.

If PPOPROC does end, an NRD message is sent to all monitor status OCS operators, and normal PPO processing resumes.

Messages from PPOPROC

Messages from PPOPROC, including those issued by &WRITE statements or resulting from commands executed from within PPOPROC, (unless &INTCMD is used) are displayed, prefixed with a P, at all monitor status terminals.

PPOPROC Statements

Special processing verbs are provided for PPOPROC use:

- &PPOREAD retrieves the next PPO message for processing
- &PPOCONT returns a PPO message for normal delivery
- &PPODEL deletes a PPO message
- &PPOREPL replaces PPO message text
- &PPOALERT generates a message to send to PPOPROC

Note: For more information about these verbs, see the *Network Control Language Reference Guide*.

Multiple occurrences of any of these statements simplifies the processing of group messages. The procedure can be structured for each type of message being processed.

In addition, PPOPROC can start worker processes to gather other relevant network information to act on PPO messages.

Test PPOPROC

The SYSPARMS PPOPROC command is used to invoke PPOPROC, and can also be used to flush it. This allows a new copy of the PPOPROC procedure to be invoked from a subsequent SYSPARMS command.

The &PPOALERT statement lets you generate test PPO messages which can be used to check the logic of PPOPROC without having to wait for a real occurrence of the message.

Note: The base PPOPROC procedure never uses a preloaded copy of the procedure, so it is unnecessary to use the UNLOAD command. Other procedures invoked from within PPOPROC will observe the normal preload conventions.

PPOPROC Prerequisites

The following prerequisites apply:

- Before your product can receive messages from VTAM, a PPO START command must be successfully executed to authorize the system for the receipt of unsolicited VTAM messages. The PPO START opens the VTAM ACB specified on the SYSPARMS PPOACBNM= to gain access to VTAM messages. The STATUS command can be used to determine whether the system has successfully executed a PPO START command.
- To start your PPOPROC, specify SYSPARMS PPOPROC=*procname*.
- If you want to receive copies of VTAM commands in your PPOPROC, you must have SYSPARMS PPOSOCMD=PPOPROC specified and verify that PPOLOG=YES is specified in your VTAM startup.
- To receive specific messages, issue DEFMSG DELIVER=PPO commands either in your PPOPROC, or before starting it.
- PPO message delivery to PPOPROC begins after the first &PPOREAD has been issued.

Note: For more information about the SYSPARMS command, see the *Reference Guide*. For more information about the PPO, STATUS, and DEFMSG commands, see the online help.

Intercept OCS Messages (MSGPROC Procedures)

Your product can intercept all messages being sent to an OCS screen or background environment.

This facility is supported by a procedure known as MSGPROC. Before this facility can be utilized, a MSGPROC must be defined for each user ID requiring message interception.

To do this, update the user ID definitions (using UAMS, or an external security system) to define the name of the MSGPROC to be invoked.

The name provided in the MSGPROC field of the user ID definition is the name of a procedure in the procedure library. Different user IDs can have different MSGPROCs, while some user IDs might have no MSGPROC at all. You can change or flush the MSGPROC value with the PROFILE command.

Note: MSGPROC is supported for OCS mode and INMC users with ROF sessions, including background user IDs and console user IDs. However, you cannot use MSGPROC in dependent processing environments associated with &INTCMD use.

MSGPROC Statements

Special processing verbs are provided for MSGPROC use:

- &MSGREAD makes the next message available.
- &MSGCONT returns a message for normal delivery.
- &MSGDEL deletes a message.
- &MSGREPL replaces the text of a message.

Note: For more information about these verbs, see the *Network Control Language Reference Guide*.

Multiple occurrences of any of these verbs simplifies the processing of group messages, as the procedure can be structured for the type of messages being processed.

In addition, MSGPROC can use the &INTCMD processing facilities to perform other functions while processing a message.

Note: An MSGPROC procedure receives all messages that appear on your OCS window. The procedure can therefore be used for various purposes, for example:

- Acting on unsolicited messages received on MAI sessions
- Reformatting VTAM messages
- Recognizing events and prompting an operator to take appropriate action

Operators with suitable authority can change the name of their MSGPROC procedure or flush it, using the PROFILE command.

Design MSGPROC Procedures

You write an MSGPROC procedure as a closed loop. When invoked, it processes until the first &MSGREAD statement is detected again and then suspends processing until a message arrives for your terminal.

As each message arrives, the procedure logic branches to an appropriate processing point for that message, or issue an &MSGCONT if the message is of no interest. After processing the message or a group of associated messages, the procedure then returns to the initial &MSGREAD where processing is suspended once again.

If the logic of the MSGPROC procedure allows it to terminate, an error message is issued and normal message processing resumes.

MSGPROC can also use the START command to create worker processes that process events or message groups asynchronously. This lets MSGPROC continue monitoring the flow of OCS window messages.

Messages from MSGPROC

If a user ID is defined with a MSGPROC, all messages destined for that user's OCS window are directed to the MSGPROC for processing. Messages that come from within the MSGPROC itself (such as those from &WRITE statements, or comments written to the user's terminal), are not passed to the MSGPROC for processing. This prevents recursive looping.

Test MSGPROC

MSGPROC is invoked when you enter OCS mode. MSGPROC processes until the first &MSGREAD statement is encountered. The procedure remains active until terminated by an error (in which case normal message delivery resumes), or until you exit OCS mode. Exiting OCS mode flushes the MSGPROC procedure.

Subsequent reentry to OCS mode invokes the latest copy of the procedure, unless the procedure has been preloaded. If the procedure has been preloaded, you first unload it using the UNLOAD command and LOAD the replacement procedure.

The system is distributed with an MSGPROC example. Before you write your own MSGPROCs, review $MSGPROC in the distribution library.

Define a temporary user ID where $MSGPROC is defined as the required MSGPROC, then log on to this user ID and enter OCS mode to invoke the MSGPROC.

$MSGPROC intercepts VTAM display commands and provides a single-line summary. The procedure extracts information from various lines generated by the VTAM command, analyzes them within the procedure, and generates a response to the operator. You can tailor the example for your VTAM level.

User ID Considerations for System Level Procedures

All NCL processes execute on behalf of an authorized user ID. Private users who log on and execute NCL processes, execute their NCL processes within their own NCL processing regions.

System level procedures also require an NCL processing region for execution. The principal procedures, LOGPROC and PPOPROC execute under special internal user ID environments. LOGPROC executes under a special user ID, and PPOPROC executes under a special PPO interface user ID.

The internal user IDs, which provide the execution environments for these system level processes, are regarded as standard users. They are logged on in the standard way and can have UAMS (or security exit) user ID definitions. If you want to profile these internal user IDs in special ways, you can define them in the same way as real users.

Chapter 10: Implementing User Programs

This section contains the following topics:

About the SUBSYS Facility

The SUBSYS facility lets you call user programs and have them run in a separate subtask. This lets you:

- Enhance the &CALL NCL verb by attaching a long-running subtask to handle &CALL requests
- Preserve the state of system resources, such as open files, external database interfaces, and so on
- Reduce the system overheads associated with each call

The SUBSYS facility is supported by the SUBSYS command, and extensions to the &CALL NCL verb.

You should not confuse the subsystems provided by the SUBSYS facility with the z/OS Subsystem Interface, or other z/OS subsystems. These are purely internal subsystems managed by your product.

Use &CALL Without SUBSYS

If you are using &CALL without the SUBSYS facility, the following disadvantages are encountered:

- An z/OS ATTACH is issued for each call. This means that the called program cannot easily remember things across calls.
- Significant CPU time is expended in ATTACH processing
- Continual opening and closing of files by z/OS
- Use of either &LOCK (in NCL) or ENQ/DEQ (in assembler) to serialize

Use &CALL With SUBSYS

To counter the disadvantages of using &CALL, you can use the SUBSYS facility. This provides the following:

- The target program is only attached once and all work is queued to it
- The target program can easily keep files open
- The target program is informed of NCL process termination, and of subsystem start and stop. It is also notified of system shutdown.

The SUBSYS program is attached once, as a long-running task so that you can open files, get storage, and so on. These resources are not released when you return.

Extensions to &CALL

The SUBSYS command lets you nominate a subsystem as a complete &CALL replacement. This means that any existing NCL procedures that have used &CALL to call a program, for example, X, can be automatically redirected to a SUBSYS defined as X. This command is one way to gain immediate improvements in performance for some programs.

Note: For more information about the &CALL verb, see the *Network Control Language Reference Guide*.

Uses of SUBSYS

Some examples of the use of the SUBSYS facility are:

- Access to foreign DBMS
- The ability to read and write non-supported file formats from NCL
- High-performance reworks of existing &CALL programs
- Interfaces to some IBM utilities such as IDCAMS. One example is UTIL0035.

Send Parameter Lists to the Subsystem

When a call is made to an NCL procedure using the SUBSYS facility, parameters are passed to that procedure. These parameters are passed in a list which can be in an old or a new format.

Note: For more information about the old parameter format, see the description of &CALL in the *Network Control Language Reference Guide*.

The new parameter list format is available for direct call-attach and is specified in the PARMLIST=NEW operand of the SUBSYS DEFINE command. The new format provides the following:

- Called programs can determine whether or not they have been called with an old or new format list.
- Provides more information about the environment. There are sections that describe the current product region, USER, and NCL processes.
- Additional NCL process termination, initialization, and shutdown calls.
- NCL process correlators which allow you to be independent of the shared NCL process correlator by writing reentrant SUBSYS code.
- Callback application program interface (API) entry point that your program can use to send messages back to the region.

The PARMLIST format is described in the distributed macro, $NMNCPL.

Note: The new format PARMLIST is compatible with high-level languages that use standard IBM linkage conventions.

Control Subsystems

In order to control your subsystems you need to be able to perform various actions on them. Each subsystem can be:

- Defined
- Started
- Stopped
- Deleted
- Reloaded
- Have its status displayed

Define a Subsystem

To define a subsystem to your product, use the SUBSYS DEFINE command. When this command is issued, the nominated subsystem is defined and the indicated program is loaded into storage. By default, the subsystem is also started. To prevent a subsystem being started when being defined, specify the NOSTART parameter.

Note: For more information about the SUBSYS DEFINE command, see the online help.

Start a Subsystem

A subsystem must be started before it can accept work. To start a subsystem, use the SUBSYS START command. This command is used to start a defined or inactive subsystem. A subsystem can also be automatically started when it is defined.

When a subsystem is started, the following processing occurs:

- A small part of your product is attached
- An initialization of the attached product code occurs
- If the subsystem has PARMLIST=NEW in effect, an initialization call is made to the subsystem code [NCPFFUNC = NCPFFSIN (8)].

The subsystem is now allowed to accept calls (&CALL statements).

Stop a Subsystem

A subsystem can be stopped. This allows it to stay defined, but calls are no longer permitted. To stop a subsystem, use the SUBSYS STOP command. When the SUBSYS STOP command is entered, the following occurs:

- If the subsystem is defined with PARMLIST=NEW, a stop call [NCPFFUNC = NCPFFSTM (12)] is made to the program. This allows the program to clean up any important resources.
- The subsystem is detached
- Any pending &CALL requests are rejected. No further &CALL requests are permitted.

A stopped subsystem can be restarted by the SUBSYS START command.

Force a Subsystem to Stop

Sometimes a subsystem hangs in the user code. This could be caused by a loop, or an unsatisfied WAIT. Regardless of the reason, all NCL processes waiting on a call may wait indefinitely. This is especially likely while developing subsystem code. To find out whether the subsystem has hung, use the SHOW SUBSYS command.

You can stop a hung subsystem with the SUBSYS FORCE command. The subsystem is force-detached, cleaned up and then assumes the STOP status.

Delete a Subsystem

A subsystem definition can be deleted by using the SUBSYS DELETE command. You must stop the subsystem before it can be deleted. Any following requests to this subsystem are rejected.

Reload the Program

If, while developing a subsystem, a new version of the program is required, then the SUBSYS RELOAD command can be used, as an alternative to STOP/DELETE/DEFINE.

The indicated subsystem must be stopped. The program is deleted (provided that it has no other users), and a new copy is requested. The subsystem can then be restarted by using the SUBSYS START command.

Note: If the program has other users, the delete does not work, and the reload obtains the same version as before.

Display the Status of a Subsystem

To display the status of all defined subsystems, use the SHOW SUBSYS command.

Write a SUBSYS Program

A SUBSYS program has to be able to handle the following events:

- An NCL process that called the subsystem terminates.
- An initialization call is made.
- A SUBSYS STOP command is executed.
- Your product starts to shut down.
- A subsystem cleanup is performed.

To make it easy to track data associated with a given environment (such as an NCL procedure), various correlator words are provided from which you can anchor control block structures.

The following sections describe how to write a SUBSYS program using the new parameter format list.

Note: For more information about how to write a program using the old parameter list, see the &CALL description in the *Network Control Language Reference Guide*.

Subsystem Program Considerations

When writing a subsystem program, the following should be considered:

- Use the new format parameter list. Remember that you can validate the format by checking that the value pointed to by R1 on entry is equal to R1.

  ```
  C      R1,0(,R1)
  BE     NEWFMT
  B      OLDFMT
  ```

  The following facilities are only available if the new parameter list format is used:

  - An NCL process cleanup call
  - System initialization calls
  - System shutdown calls
  - Private correlator calls
  - Callback API

- Check whether your program has been called as a subsystem rather than an &CALL attach. If an asterisk is found in the subsystem name field in the (new) parameter list, then it is an &CALL attach, not a SUBSYS call.

- A subsystem gets a private correlator word for each unique NCL process that calls it. This correlator word can be altered (it starts as 0) and is remembered and returned on the next call for that NCL process. It is an ideal place to anchor process-related control blocks. The shared correlator is shared with all subsystems or other &CALLed programs for this NCL process.

- Any NCL process that calls a subsystem using the new parameter list causes a cleanup call to be provided to that subsystem. This call is made regardless of the type of NCL process termination (normal, abnormal, and so on). The private correlator is provided, and this allows the subsystem to clean up any control blocks as required. If an NCL process is flushed while on a SUBSYS call, the subsystem is not notified of the fact until later. It need not worry about the process disappearing while actually processing the current call.

- A SUBSYS DEFINE (without NOSTART) or SUBSYS START command sends an initialization call to the subsystem. This allows the subsystem to initialize its environment. It should anchor any control blocks in the SUBSYS correlator in the new format parameter list.
- A SUBSYS STOP sends a termination call to the subsystem program. This allows the subsystem to clean up any control blocks, and so on, prior to being detached.
- If your product is shut down, a system shutdown call is sent to the subsystem. It must quickly clean up its environment. It is force-detached several seconds later regardless of whether it is finished or not.
- Only as many parameters as were passed can be returned to a caller. This is the same as the previous &CALL. Each parameter is limited to 256 characters of data. Binary values are preserved both on input and on return. Be sure to set the lengths correctly.
- All calls to a specific subsystem are queued and processed in turn. This means that, if the subsystem code takes a long time to process a call (for example, issuing a WAIT on VTAM input), all pending calls are also delayed. This is a restriction of the current implementation.
- This can be avoided by queuing requests to the subsystem to a manager process (using the INTQ command and a globally known NCLID). This manager process talks to the subsystem on behalf of all users of the subsystem. Then the subsystem code can wait on many events and return the first event to the manager, which can then pass it back to the requestor.
- If the subsystem ABENDs, the subsystem will stop. It can be restarted.
- SUBSYS FORCE is useful when a subsystem hangs.
- The subsystem load module must reside in a library accessible to your product region. Typically this is the STEPLIB. Remember that your product libraries must be authorized. This means that any libraries in this STEPLIB concatenation must also be authorized. This is a security consideration.
- 31-bit mode programs are supported. All supplied control blocks (for example PARMLIST) are always below the 16mb line.
- If performing multiple functions, use the first parameter passed, as a function indicator:

```
&CALL SUBSYS=SUB1 OPEN ....
&CALL SUBSYS=SUB1 READ ....
&CALL SUBSYS=SUB1 READ ....
&CALL SUBSYS=SUB1 CLOSE ...
```

Write a Subsystem Program in High-level Languages

The new format parameter list makes it possible to write both one-time programs using the &CALL command and subsystems in high-level languages, for example COBOL and PL/1.

There are several considerations for high-level languages:

- Always use a new format parameter list. The old format is incompatible with high-level languages.
- In COBOL, define the input parameters in the LINKAGE SECTION. COBOL cannot easily handle a variable number of parameters, but as long as the extra parameters on a specific call are not referred to (that is, the count is obeyed) then no problems should arise. Another technique may be used to access a variable number of parameters. The individual parameters are actually in contiguous storage. Thus, instead of individual parameters, just define the first one, as an array:

  ```
  LINKAGE SECTION.
  ... other parms
  01 NCPC.

  03 NCPCCNTPIC S9(9) COMP.
  01 PARMS OCCURS 1.  ... actual count is NCPCCNT
  03 LENPIC S9(9) COMP.
  03 VALPIC X(256).

  PROCEDURE DIVISION USING PLIST ... ... NCPC PARMS.
  ```

- Dope vectors are not set up for PL/1. Because of this, define all individual input parameters as FIXED BIN(31) and BASE a structure over them:

  ```
  DCL NCPN_           FIXED BIN(31),
      1  NCPNBASED     ADDR(NCPN_F),
        3  NCPNPROC    CHAR(8),
  ```

  The suggestion for COBOL regarding variable parameter counts applies to PL/1 also. Since the actual number of parameters is not part of the pseudo-array, a REFER structure cannot be used. Use PARMS(1) on the DCL.

  Do not compile with SUBSCRIPTRANGE in effect.

- Use the RETURN-CODE variable in COBOL to set the return code. Use CALL PLIRETC (code) IN PL/1.
- Use GOBACK rather than STOP RUN in COBOL to return.
- Both COBOL and PL/1 have large overheads for establishing the run-time environment. When repeated calls are necessary, these overheads can be excessive. Consult the relevant programmer's guide for details on callable interfaces that can perform a one-time build of this environment.

SUBSYS Callback API

If the subsystem is using the new format parameter list, then a callback API is available. You can call this API from a SUBSYS program, including any subtasks of that program, to perform several services. (For an example of the use of this API in a subtask, see the source of the UTIL0035 program.)

The address of the API entry point is in the NCPSAPIE field, as defined by the $NMNCPL macro. The address in this field is different for each active subsystem and does not change during the life of that subsystem. You can save it elsewhere (for example, for use by subtasks), but must not share it across subsystems or across subsystem restarts.

After loading the entry point into Register 15 (R15), you can call the API by a standard BALR R14,R15 instruction. The caller can be in any AMODE, but all addresses passed in the parameter lists must be valid 31-bit addresses.

The API expects a standard format parameter list. Depending on the request, the list contains three or four parameters.

Register 1 must point to a list of addresses, each address in turn pointing to a parameter. The end-of-list bit is not required and not checked.

The first parameter is a fullword binary function code: 0, 4, and 8, which are described in the following sections.

Function Code 0—Queue a Message to the INTCMD Environment of an NCL Process

Function Code 0 queues a message to the INTCMD environment (response queue) of a nominated NCL process. Three additional parameters follow the function code:

parameter_2

Specifies the length of the message in fullword binary format.

Limits: 1 to 256

parameter_3

Specifies the message to queue.

parameter_4

Specifies the NCL ID of the NCL process in fullword binary format.

Limits: 1 to 99999

The process must have issued a request to the subsystem; otherwise, the message is discarded.

Important! If the message is discarded, the caller does not receive any indication.

Function Code 4—Send a Message to an NCL Process

Function Code 2 sends a message to the standard message environment of a nominated NCL process. Three additional parameters follow the function code:

parameter_2

Specifies the length of the message in fullword binary format.

Limits: 1 to 256

parameter_3

Specifies the message to send.

parameter_4

Specifies the NCL ID of the NCL process in fullword binary format.

Limits: 1 to 99999

The process must have issued a request to the subsystem; otherwise, the message is discarded.

Important! If the message is discarded, the caller does not receive any indication.

Function Code 8—Send a Message to MONITOR Receivers

Function Code 8 sends a message to all MONITOR receivers in the region. Two additional parameters follow the function code:

parameter_2

Specifies the length of the message in fullword binary format.

Limits: 1 to 256.

parameter_3

Specifies the message to send.

Return Codes

Following a call to the API, R15 is set to one of the following values:

0

Indicates that the function works.

4

Indicates that the specified length of the message is invalid.

8

Indicates that the specified NCL ID is invalid.

24

Indicates that the specified function code is not supported.

28

Indicates storage shortage.

32

Indicates that the caller is not executing as a subsystem.

Example: Send a Message

This example shows how a message can be sent to the current caller's environment. (It is assumed that an &CALL request is being processed by the subsystem code, and that addressability to the NCPS and NCPN areas has been set up.)

```
      MVC   CBFC,=F'4'                  FUNC CODE
      MVC   CBML,=A(L'MSG1)             MSG LENGTH
      L     R15,NCPSAPIE                GET API EPA
      CALL  (15),                       CALL API          *
            (CBFC,CBML,MSG1,NCPSNCLI),  WITH THESE PARMS  *
            MF=(E,CWA)                  PLIST BUILT HERE
      ...
MSG1  DC    C'HELLO FROM A SUBSYSTEM'
      ...
CWA   DS    4F                          PARMLIST BUILT HERE
CBFC  DS    F                           FUNCTION CODE
CBML  DS    F                           MESSAGE LENGTH
```

Chapter 11: Synchronizing Access to Resources

This section contains the following topics:

Use NCL to Synchronize Access to Resources

NCL applications involving multiple users often need a mechanism for controlling access to the same data or resource, by different users or processes. This control is particularly important in NCL systems that use UDBs to hold application-related data that can be updated by some users, and concurrently read by others.

NCL provides this mechanism through the use of locks which allow or deny access to resources. The same mechanism is also used to synchronize activity between separate processes by providing a semaphore capability.

Resources and Resource Locks

A resource, as used here, is not a real entity such as a file or a variable. It is a name, consisting of a primary name and optionally a minor name, to which NCL procedures refer.

The purpose of the lock mechanism is to provide assurance that a specific operation can be performed, at a particular point within an NCL procedure's logic, without interference or damage by other NCL processes that might be attempting to use the same data at the same time.

A common example is the case where a process needs to update a record on a UDB. The process reads the target record, changes it and writes it back to the file, but it also needs to guarantee that no other process updates the same record at the same time. To achieve this guarantee, the process first obtains exclusive access to a resource that symbolizes the record update process.

Once it has obtained this exclusive access, the procedure is free to perform as much work on the target record as it needs, knowing that no other procedure can access the same record in the meantime because no other procedure would be able to gain access to the resource lock.

It is important to remember that it is not the data itself that is protected by the lock mechanism, only the resource. If controlled access to an item of data is required then all procedures that change that data must gain access to the lock before updating the data.

Resource Groups

To explain the concept of a resource group, take the previous example of a file record, in which a procedure uses resource locking to guarantee exclusive access to a file record for the purpose of updating it, without having to worry about any other process changing the record at the same time. In this example, the resource could be a name that represents the entire file; in other words, you could organize your procedure to have exclusive read/write access to a whole database.

Alternatively, and probably in preference, you would like other processes to continue to have access to the file as a whole, as long as they could not access the specific record that your process is updating. To achieve this, you would assign a resource primary name to represent the file (UDB), and then decide on a naming convention that allows individual records within the file to be identified by a minor name.

For example, if you have a UDB containing NCP configuration data keyed by line name, you might assign a primary name of CONFIG to represent the UDB itself and use the line name as a minor name to represent each line record as a resource within the UDB. To retrieve information about a line, your procedure first obtains exclusive access to the appropriate line record resource by requesting exclusive control of the appropriate primary/minor name resource lock.

In this example, your procedure would execute the following &LOCK statement to gain exclusive permission to process the record on the UDB that contains information about line 23:

```
&LOCK TYPE=EXCL PNAME=CONFIG MNAME=LINE23
```

You could code the following statement if your naming convention for resource locks uses XYZ to identify the record on the database that describes the configuration of line 23:

```
&LOCK TYPE=EXCL PNAME=CONFIG MNAME=XYZ
```

In this example, the primary name (CONFIG) represents a resource group. The combination of the primary name and a minor name identifies a resource within the group.

Primary Names

The primary name is the part of the resource name that uniquely identifies the resource group. It is a 1- to 16-character string of your choice. Logically, the primary name represents the root of a (potential) two-level hierarchy, below which one or more dependent minor names can exist.

Minor Names

The minor name of a resource, if required, is a 1- to 256-character string, which qualifies the resource's primary name to identify a specific resource within a resource group. For example, the primary name CONFIG represents both a resource group and a specific resource. The resources CONFIG.ABC and CONFIG.XYZ represent specific resources within the CONFIG resource group.

Resource Name Hierarchy

Specific resource names, that is, those that are explicitly defined by primary and minor name, are peers within their resource group.

Any process that obtains exclusive control of the resource CONFIG prevents any other process in the system from gaining access to any other resource below CONFIG in the hierarchy. Therefore no process can gain access to the CONFIG.XYZ resource, until the first process releases its exclusive control of the CONFIG resource lock.

Alternatively, a process that requests exclusive control of the CONFIG.ABC resource does not prevent any other process from accessing the CONFIG.XYZ resource.

Resource Naming Conventions

Since resources are only names, not real entities, the NCL procedures that use the locking mechanism to control and synchronize their access to different resources must agree on the resource names that they will use. They must also agree that a resource name means the same thing to all procedures. Once the resource names have been agreed, all procedures must use the &LOCK verb to obtain the resource lock to access resources, otherwise protection cannot be guaranteed.

It is very important that you define a naming convention for resource identification. Ideally your naming convention should apply to all product regions within your organization. Alternatively, naming conventions can apply within a given system or within a specific NCL system.

&LOCK Verb

The &LOCK verb lets you obtain and release resource locks.

- &LOCK operates systemwide; it is not restricted to your NCL region alone.
- &LOCK is used to designate a particular resource lock that the procedure wants to control, and specifies the type of control that is required.

If a process has exclusive control of a resource lock, all other processes in the system are prevented from gaining access to the resource at the same time. Alternatively, the process can gain shared access to the resource lock, which prevents any other procedure from being granted exclusive access. If necessary, you can alter the status of the lock from shared to exclusive, or from exclusive to shared, during processing.

A procedure can also use &LOCK to test whether any other procedure is holding a lock that it wants to access.

Note: For more information about the &LOCK verb, see the *Network Control Language Reference Guide*.

Wait for Access to a Resource

If a procedure issues &LOCK to request access to a resource lock, but the required access cannot be granted immediately, the WAIT operand specifies whether the system will return control to the procedure immediately, or wait for the resource to become available.

If you specify WAIT=YES, your procedure will be suspended indefinitely until access to the required resource can be obtained. You should avoid this, unless you are certain that a deadlock condition with another process in the system will not result.

Rather than code WAIT=YES, CA recommends that you use the WAIT=nnnn option. This instructs the system to wait for the required resource for a certain number of seconds (as specified by the nnnn variable), rather than waiting indefinitely. If the lock is still not available after that time, the procedure resumes processing. This stops deadlock conditions occurring.

Alter the Status of a Resource Lock

If required, you can alter the resource lock status of a process during processing by using the ALTER=YES operand on the &LOCK verb. Altering the lock status from exclusive to shared is always possible, however there are some restrictions when altering the lock status from shared to exclusive.

Alter the Status from EXCL to SHR

During processing of the &LOCK request, any other lock requests that are waiting for shared access to the resource become valid for shared ownership. They are granted shared access to the resource immediately, causing the requesting procedures to resume execution.

Alter the Status from SHR to EXCL

The following conditions must be satisfied before a request to alter a resource lock status from shared to exclusive will be successful:

- No other procedures can have shared ownership of the resource
- If the resource is the primary resource, there must be no other minor resources (with shared or exclusive status) with the same primary name

If any other shared requests for the lock arrive before the status is altered to exclusive, these new shared requests are given precedence over the change to exclusive, and are granted shared ownership of the lock

If the request is successful the procedure owns the lock exclusively.

WAIT Operand

As with normal shared and exclusive requests, the WAIT operand plays an important part in determining the success or failure of a request to alter the lock status. However, the waiting period is only significant for a request to change from shared to exclusive, as the request to change from exclusive to shared is always satisfied immediately.

When WAIT=NO is specified, the request fails unless it is satisfied immediately.

When WAIT=nnnn or WAIT=YES is specified, the requesting process can wait for the specified period of time for the change to be successful (this will happen when another process releases its lock).

If the status type on the &LOCK ALTER request is the same as the current status, the request is treated as a request to alter the lock text only-for example, the process holds a shared lock then issues a shared lock request with the ALTER=YES operand. This request is always successful.

Associate Text with a Resource Lock

To assist recognition or provide information to other processes that want to interrogate the status of a lock, the &LOCK verb can specify text associated with your procedure's ownership of the lock. This text appears in the SHOW NCLLOCKS command display, which is used to display the locks held by the different processes in the system. The text is also made available to a procedure issuing an &LOCK statement with the TYPE=TEST option.

Note: For more information about the SHOW NCLLOCKS command and its use, see the online help.

Resources as Semaphores

Using &LOCK to synchronize the use of a resource between competing processes solves the problem of access to real resources. The same technique can be employed to signal between co-operating processes, in particular with the use of the TEST facility of &LOCK. In this case, the resource is not a real resource but a signal or semaphore.

Semaphores are a useful mechanism for synchronizing the processing of related procedures. Take an example of Procedure A that needs to suspend processing until Procedure B reaches a particular point in its processing.

Procedure A's logic could look like this:

```
.
.  -* process
.
&LOCK WAIT=YES TYPE=TEST PNAME=XYZ
```

At this point, Procedure A is suspended until another &LOCK request is made for the resource XYZ.

Procedure B in the meantime continues processing until it completes the function which it knows procedure A has to wait for. At this point, procedure B issues the same &LOCK request for the resource XYZ:

```
&LOCK WAIT=YES TYPE=TEST PNAME=XYZ
```

Both the &LOCK verbs complete with &RETCODE = 8, indicating that both procedures have reached the synchronization point. If Procedure B reaches the point first, its &LOCK would suspend it until Procedure A issued its &LOCK statement.

Chapter 12: NCL Debug Facility

This section contains the following topics:

Overview

The NCL Debug facility is a powerful tool to assist in the debugging of NCL procedures.

The DEBUG command has various operands which provide the NCL developer with the ability to establish a debug session. This debug session can be targeted at one or more debug scopes. A debug scope can be any of the following:

- An NCL process
- All the processes within a particular user's window
- All the processes within a particular user's logon region
- All the processes for a particular user ID (that is, all regions for that user ID)

You can have one or more debug sessions active simultaneously, each being initiated from a different environment.

A debug session is associated with a particular region that is referred to as the debugger. This region is the only region from which debug commands are accepted for the session. Any command environment can be a debugger. These environments include OCS and any &INTCMD environments.

An NCL process that falls into the scope of a debugger is associated with that debugger's environment and is referred to as a debugged NCL process. Only one debugger can debug an NCL process at a time.

After a debug session has been started, you can set break points in any procedure of any NCL process that has been attached to the debug session. The breakpoints can be set at particular points within the executed code, against the action of updating a variable, or against the execution of a nominated verb. The debugger can view or alter the data of the target NCL process. These breakpoints are external to the source and allow you to debug a process without having to modify the source code.

You can control execution of an NCL process by setting various breakpoints throughout the process. The DEBUG commands allow you to halt the execution of a process, resume execution (after a halt or breakpoint has been reached) or to step through a fixed number of process statements. You can also suspend the debug session and reestablish it at a later time. Any breakpoints or suspended processes is preserved in their current state.

The SHOW DEBUG command lets you see all your current debug sessions and their associated scopes and, optionally, the debug sessions of other users.

Note: For more information about the DEBUG and SHOW DEBUG commands, see the online help.

Security

A set of security constraints is incorporated with this flexible debugging interface, to protect against intentional access to user's NCL processes that are running.

- Command authority level can be used to stop a user from debugging the procedures of another user.
- NCL Debug is a feature of your product and therefore can be excluded from a product region using an EXC= JCL parameter.

  Note: For more information about using the EXC= parameter, see the *Reference Guide*.
- You can replace the DEBUG command by an NCL procedure using the SYSPARMS CMDREPL operand.

  Note: For more information, see the *Reference Guide*.
- The DEBUG DISPLAY command can be used to display the contents of variables but it does not display the &USERPW variable.

NCL Debug Facilities

NCL Debug provides the following facilities:

- Allows observation of the execution of an NCL procedure from an external source, that is, another environment, another user region, window, or NCL environment.
- Eliminates the need for code changes to debug a procedure. For example, there is no need to add &CONTROL or &WRITE statements.
- Provides comprehensive control over the NCL procedure as it is being executed, supporting statement stepping, alteration of variable contents and attributes, and so on.
- Allows specification of criteria for debugging, before the target NCL begins execution.

NCL Debug Facility

The NCL Debug facility is made up of a set of commands that allow you to do the following:

- Start and stop an NCL debug session
- Control the execution of NCL processes
- Display and modify the contents of NCL variables, independent of the nesting level of the process
- List the procedure and subroutine nesting levels
- Display the source code that is being executed (not the source currently on disk, that is, without comments or indentations)
- Receive NCL trace output at another region, thus being able to view the trace output concurrently as the debugged process executes

Start and Stop an NCL Debug Session

An NCL debug session is started in an environment by issuing the first DEBUG START command. The environment and scope of the session are defined using the operands of the DEBUG START command. Subsequent DEBUG START commands add additional scopes to the debug environment already established.

When the DEBUG START is processed, any NCL processes that fall within the specified scope, and which are not already debugged, will be attached to the debugger's session. If an NCL process starts, and its environment is being debugged, or it falls within the scope of a debug session, it will be attached to that debug session.

Once an environment has established a debug session, other debug commands can be issued to control the NCL processes that have been attached. The commands to manipulate these processes must be issued from the same environment that issued the DEBUG START. The process has no awareness of being debugged.

Once debugging is complete, the debug session can be terminated using the DEBUG STOP command. There are three options for the DEBUG STOP command that affect the actual processes that have been attached to the debugger.

- The CONTINUE option tells debug to remove all breakpoints from the processes and resume any that are suspended, leaving them to continue as if no debugging had taken place.
- The FLUSH option causes all attached processes to be flushed.
- The SUSPEND option leaves the debug environment intact and disconnects it from the debugger's environment (the debug session is suspended).

The CONTINUE and FLUSH options both result in the debug session being deleted. All breakpoints and scopes are cleared and any new processes that start, and fall within the scope, will not be attached.

The SUSPEND option leaves the debug environment intact. Breakpoints and scopes remain in effect. If a process starts within a scope, that process will be attached to the debug environment and any breakpoints that are pertinent will be applied. The debug session will be flagged as suspended, and a debug ID will be assigned to it. When the debug session needs to be reconnected, a DEBUG START command can be issued specifying the DEBUGID that was assigned to the suspended debug session.

A debug session can be reconnected to another environment of the same user ID. Conversely, a session can be reconnected from another environment of the same user ID. During the time that the session was suspended, new processes can be attached from the debug session. If an attached process encounters a breakpoint, it is suspended, and remains suspended until the debug session is reconnected and a command issued to resume processing for the process. The same applies for a process that was suspended at the time the debug session was suspended.

The following sequence of commands shows how to establish and stop a debug session:

```
DEBUG START WINDOW=2 -* debug any procedure in
                     -* window 2
DEBUG START USER=NM1BSYS PROCEDURE=MYBPROC
                     -* debug background procedure
                     -* in BSYS region
-* other debug commands control the process in the two scopes
DEBUG STOP TYPE=FLUSH-* terminate the debug
                     -* session and flush the
                     -* procedures
```

Control the Execution of NCL Processes

Once a debug session has been established, the debugger can issue various commands to control the execution of the processes that are subsequently attached to the debug session. The commands that can be used have the effect of either suspending a process, resuming the execution of a process, or both.

The following commands can be used to suspend an NCL process while it is executing:

DEBUG HOLD

This command flags the process for immediate suspension, before the next statement is executed.

DEBUG STEP

This command indicates to debug that the process is to be suspended after the specified number of statements have been executed. The NEXT= operand can be used to indicate how many statements of the process to allow before the process is suspended.

Both commands have the effect of putting the NCL process into a wait state. The SHOW NCL command indicates that the process has been suspended.

Another way for a process to become suspended is indirectly, using the DEBUG BREAKPOINT command. The DEBUG BREAKPOINT command defines a condition that must be satisfied before the process is suspended. The following conditions can be specified:

- Statement
- Verb
- Variable
- Procedure ENTRY
- Procedure EXIT

Statement Breakpoints

A statement breakpoint can be used when the process is to be suspended if the statement identified by the STMT= operand of the DEBUG BREAKPOINT command is about to be executed. The statement is identified using the line number found in columns 73 to 80 of the source, or a relative number if the source is unnumbered.

Verb Breakpoints

A verb breakpoint can be used when the process is to be suspended if the statement to be executed contains the verb identified by the VERB= operand of the DEBUG BREAKPOINT command. The process is suspended immediately before the verb is executed.

Variable Breakpoints

A variable breakpoint can be used when the process is to be suspended if a variable, identified by the VARS= operand of the DEBUG BREAKPOINT command, is updated. The process is suspended immediately after the statement that caused the variable to be updated. Optionally, the command can specify the DATA= operand to limit the breakpoint to updates of the variables with a particular value.

Procedure ENTRY Breakpoints

A procedure ENTRY breakpoint can be used to suspend execution of a procedure before the first statement of the procedure is executed.

Procedure EXIT Breakpoints

A procedure EXIT breakpoint can be used to suspend execution of a procedure after the last statement of the procedure has been executed. This includes any normal termination statements which end the procedure (for example, &END or &RETURN).

BREAKPOINT Command

Breakpoint definitions are associated with the debug session, and are applied to procedures both when they are executed and when the breakpoint is defined. Thus, breakpoints can be defined before any processes have been attached to the debug session. When a process issues an EXEC command, the procedure being executed has any relevant breakpoints applied.

Statement breakpoints require a privately loaded copy of the procedure. (This is not necessary for the other types of breakpoint.) If the procedure is the target of a statement breakpoint, the debugger will automatically request that a procedure be privately loaded. Otherwise, normal procedure loading will remain in effect, as controlled by the SYSPARMS NCLTEST or PROFILE command.

Once breakpoints have been established, any NCL process that satisfies the conditions of a breakpoint is suspended. Breakpoints can be listed and cleared using the DEBUG LIST BREAKPOINTS command and the DEBUG CLEAR respectively. These commands are useful for controlling processes that have already started.

If you need to stop a process on the first statement, use the DEBUG SET NEWHOLD=YES profile command. Once set, it indicates to debug that all processes that are attached to this debug session are to be immediately suspended before the execution of the first statement.

When a process is suspended, other debug commands can be used to display variables and the source code.

To resume the execution of the process after all the required information has been displayed, use the following commands:

DEBUG RESUME

RESUME flags the process identified as ready to continue execution. The process is made eligible for execution and the statement step counter, set by the DEBUG STEP command, is reset.

DEBUG STEP

The process is resumed in the same way as with the DEBUG RESUME command and the statement step counter is set to the value specified on the NEXT= operand. This will result in the process again being suspended after the specified number of statements have executed.

Sample Debug Session

The following sequence of commands shows how to set breakpoints and control the execution of a process:

```
-* The debug session has already been established
DEBUG SET NEWHOLD=YES     -* Ensure the procedure is stopped
                          -* on the first statement
DEBUG BREAKPOINT PROCEDURE=MYPROC STMT=1250000
                          -* Suspend the procedure if it
                          -* tries to execute statement 1250000
DEBUG BREAKPOINT PROCEDURE=MYPROC VERB=APPC
                         -* Suspend background procedure on
                         -* first APPC verb
DEBUG BREAKPOINT PROCEDURE=MYPROC VARS=WKTRANID DATA=CH22
                         -* Suspend the procedure when it sets
                         -* WKTRANID to the data that causes
                         -* the error.  The procedure MYPROC is
                         -* started in window 2 and is
                         -* immediately suspended

DEBUG STEP NEXT=10       -* Execute the first 10 statements
                         -* Using the display commands the procedure is
                         -* verified as to the current state of its variables
DEBUG RESUME             -* Let process continue
                         -* until first breakpoint
                         -* The procedure is suspended on the APPC verb. Using the
                         -* display commands, the procedure is verified as to the
                         -* current state of its variables. They are set correctly
DEBUG LIST BREAKPOINTS    -* List all current breakpoints
DEBUG CLEAR BREAKPOINT=2 -* Clear the breakpoint on the
                          -* APPC verb
DEBUG RESUME              -* Let the process continue until
                          -* next breakpoint
                          -* Process continues
```

Display and Modify the Contents of NCL Variables

Once the procedure has reached a particular point in its processing and it has been suspended, the DEBUG DISPLAY and DEBUG MODIFY commands can be used to display and modify the contents of variables and MDOs. The entire contents of the variable or MDO are displayed.

The modify command can be used to alter the attributes and contents of variables and MDOs. Multiple variables can be altered by specifying the operands of the modify in a similar fashion to using the &ASSIGN verb, however the new value can only be specified using the DATA= operand.

List Procedure and Subroutine Nesting Levels

The DEBUG TRACE command is used to list all the procedures. Subroutine nesting levels can also be obtained. The listing shows a general display of the identified process, and then a detailed list of each procedure and the subroutines that the procedure has entered.

Display the Executed Source

The DEBUG SOURCE command lets you list the statements of a procedure, as they are stored in memory. The listing represents what the system will actually execute, as opposed to what is currently on disk. This can be useful in verifying that the procedure being debugged is the correct version.

Receive NCL Trace Output

When debugging full screen procedures, the output from the NCLTRACE facility is not available for viewing until the process has completed, or the window has been released by the process. This can be undesirable if the process has been suspended after a panel has been sent and the process still owns the window. The trace output will not be seen until the process releases ownership of the window.

Using the DEBUG SET NCLTRACE=YES profile command, the trace output from a process being debugged will be delivered to the debugger's environment.

Example: Debug Session

The following sequence of commands shows how a debug session could proceed:

```
DEBUG START WINDOW=2          -* Debug any procedures in
                              -* window 2
DEBUG START USER=NM1BSYS PROCEDURE=MYPROC
                              -* Debug background procedure
                              -* MYPROC in BSYS region
DEBUG SET NEWHOLD=YES         -* Ensure procedure is
                              -* stopped on 1st statement
DEBUG BREAKPOINT PROCEDURE=MYPROC STMT=1250000
                              -* Suspend the procedure if it
                              -* tries to execute statement
                              -* number 1250000
DEBUG BREAKPOINT PROCEDURE=MYPROC VERB=APPC
                              -* Suspend background
                              -* procedure on the first
                              -* APPC verb
```

```
DEBUG BREAKPOINT PROCEDURE=MYPROC VARS=WKTRANID DATA=CH22
                              -* Suspend the procedure when
                              -* is sets WKTRANID to the
                              -* data that causes the error
                              -* The procedure MYPROC is started in window 2 and
                              -* is immediately suspended
DEBUG STEP NEXT=10            -* Execute the first 10
                              -* statements
DEBUG DISPLAY VARS=WK* GENERIC
                              -* Verify that the work
                              -* variables have been set
                              -* correctly
DEBUG RESUME-* Let the process continue
                              -* until the first breakpoint
                              -* The procedure is suspended on the APPC verb
DEBUG DISPLAY VARS=WKTRANID   -* The variable has the
                              -* correct value - continue
DEBUG LIST BREAKPOINTS        -* List the current
                              -* breakpoints
DEBUG CLEAR BREAKPOINT=2      -* Clear the breakpoint on
                              -* the APPC verb
DEBUG RESUME                  -* Let the process continue
                              -* until the next breakpoint
                              -* The procedure is suspended after updating WKTRANID
to
                              -* CH22
DEBUG DISPLAY VARS=WK* GENERIC-* The display indicates that
                              -* WKTRANPREF was incorrect
DEBUG MODIFY VARS=WKTRANPREF DATA=ZCH
DEBUG MODIFY VARS=WKTRANID DATA=ZCH22
                              -* Fix the variables
DEBUG CLEAR                   -* Clear all the breakpoints
DEBUG BREAKPOINT VARS=WKTRANPREF
                              -* Stop the procedure when
                              -* WKTRANPREF is updated
DEBUG RESUME                  -* Let the process continue
                              -* until the next breakpoint
                              -* The procedure is suspended after updating
WKTRANPREF
DEBUG TRACE                   -* Display the nesting
                              -* levels.
                              -* The display indicates that a subroutine was
                              -* called at the wrong place
DEBUG MODIFY VARS=WKAPPPPLCTR DATA=2
                              -* Change the loop counter to
                              -* go through the process again
```

```
NCLTRACE ON ID=123-* Set tracing on
DEBUG SET NCLTRACE=YES-* Have the trace output sent
                              -* here
DEBUG STEP NEXT=80-* Check the results
                              -* After seeing the trace, it is clear that the
                              -* variable WKHSGT is incorrectly set. The source is
amended.
                              -* The bug had been found, so stop the debug
                              -* session and flush the process.
DEBUG STOP TYPE=FLUSH         -* Terminate the debug
                              -* session and flush the
                              -* procedures
```

Chapter 13: About Mapping Services

This section contains the following topics:

What Is Mapping Services?

Mapping Services is a facility that provides NCL with enhanced capabilities for manipulating data with various formats.

Early versions of NCL stored program data only as tokens (or NCL variables). Mapping Services enhances NCL by introducing Mapped Data Objects (MDOs) as an alternative to tokens and as a means of storing data. Data can easily be transferred between MDOs and tokens. MDOs can also be written directly to files and are supported by a number of specialized NCL functions such as &ASSIGN and &APPC. MDOs can be used in many situations where tokens are used.

An MDO can contain any data that can be represented as a continuous string of bytes in storage. The primary advantage of an MDO over NCL tokens is that particular substructures within an MDO can be located and referenced by name from NCL, using various rules which Mapping Services understands. Various segments within these substructures can also be referenced. The substructures which can be referenced within MDO data are generically known as components.

To distinguish between various substructures which exist within a particular MDO, a unique tag or key is assigned to each component. To determine what names are to be associated with particular components within an MDO, and their tag values, a map is used. A map enables the NCL language to associate names with the various components which exist within a particular MDO. Thus a map provides NCL with the ability to interpret the data contained within an MDO.

Mapping Services Processing

Mapping Services extends the flexibility of NCL to include the manipulation of data in any format. There are three key components involved in Mapping Services processing:

- MDOs
- Maps
- NCL procedures

Mapping Services provides a set of facilities that effectively mediates between these three components and manages them. This allows components to exist as separate entities, which interact during NCL processing to perform data manipulation.

MDOs

Mapping Services can operate on any data item that can be represented as a continuous string of bytes in storage. In addition, Mapping Services understands certain rules which let it locate substructures within those data items if they are composed of variable data structures.

To assist with uniformity of processing, Mapping Services treats the entire data item as a structure called an MDO. An MDO has a name that is supplied by the NCL procedure to reference that instance of data.

Maps

Maps are defined using the Abstract Syntax Notation One (ASN.1) language. They are then compiled and loaded for use in NCL.

Note: For more information about defining maps, see the *Managed Object Development Services Guide*.

NCL Procedures

NCL procedures are written to access data through one or more of the standard NCL verbs. When data is provided by one of the verbs that support Mapping Services, the procedure can request that it be treated as an MDO. It provides a name for the MDO, and (optionally) the name of a map that can be used to interpret the data.

Once the data has been internally accessed, and before the NCL verb completes, Mapping Services makes a connection between the MDO and the designated map. The NCL procedure can then reference data components within the MDO using the symbolic names defined in the map.

Example: NCL Procedure

The following example shows how the three components interact in NCL.

```
&ASSIGN MDO=file1rec MAP=file1map
                    -* the map called file1map is attached
                    -* to the MDO called file1rec.
                    -* This statement indicates to Mapping
                    -* Services that this map is to be used
                    -* to interpret any data in the MDO.

&FILE OPEN ID=FILE1 FORMAT=MAPPED MAP=file1map
                    -* Open file in mapped processing mode.

&FILE GET ID=FILE1 KEY='00000000' MDO=file1rec
                    -* Read a record from the
                    -* file into the MDO.

&ASSIGN VARS=A FROM MDO=file1rec.luname
                    -* Locate a component within the MDO
                    -* data and copy its contents into the
                    -* variable &A.
                    -* Mapping Services uses the map to find
                    -* out how to locate and recognize the
                    -* structure referred to as luname-* within NCL.
```

Mapping Concepts

A map effectively describes the following to NCL:

- The names that will be used to reference various components within an MDO
- The relationship between these data components
- The means of identifying each component
- The way in which the data is represented

Until a map is associated with an MDO, NCL cannot reference components in the data by name. However, NCL can still process the MDO as a whole.

Data Sources

NCL processing can move data directly into an MDO using many verbs, via the following sources:

- From files
- Across APPC transactions
- From NCL tokens (as variables and arguments, or args)
- Across PPI transactions
- Using &INTREAD, &LOGREAD, &MSGREAD, &PPOREAD
- From vartables
- From CNM
- Using BER encode/decode

Naming

It is possible to reference structures within MDO data by name from NCL. A map must be attached to the MDO so NCL can use this map to associate names with physical structures which can occur within the data. By definition, components are hierarchically arranged. This means that to reference one component enclosed within another, a concatenation of the enclosing structure names is required to identify the enclosed one.

For example, the name:

```
MDO=FILE1REC.DOMAIN.SUBDOMAIN.LU
```

might be used to reference the structure LU, within a structure called SUBDOMAIN, which is contained in a structure called DOMAIN, which is within the data in an MDO called FILE1REC.

To reference all data contained within an MDO, only specify the single MDO name segment, as follows:

```
MDO=FILE1REC
```

Transfer MDOs Between Nested NCL Procedures

It is possible to share an MDO between nested procedures by using SHRVARS options. Any MDO selected by the current SHRVARS option is shared in the same manner that NCL tokens are shared.

Example: Transfer MDOs Between Nested NCL Procedures

In the following example, the MDO called ABC is available in the nested procedure but the MDO called XYZ is not.

```
&ASSIGN MDO=ABC MAP=MAP1 DATA=xxxx
                -* Create an MDO called ABC, attach the
                -* map called MAP1 to it, and assign
                -* data into it.
&ASSIGN MDO=XYZ MAP=MAP2 DATA=yyyy
                -* Create an MDO called XYZ
&CONTROL SHRVARS=(A)
                -* Set SHRVARS option.
-EXEC PROC2-* Call procedure PROC2.
```

Note: There is no support for passing an MDO as an invoked or returned parameter on nested calls. Only NCL tokens can be passed in this manner. However, MDOs can be transferred from one NCL process to another using &WRITE and &INTREAD or any of a number of other verbs.

Mapping Services, Mapping Support, and NCL Processing

Before structures can be referenced within an MDO by name from NCL, a map must be assigned to the MDO. The map provides NCL with the following information about an MDO:

- The type of structures which can be expected within the MDO data
- How to locate and recognize these structures
- The names which will be used in NCL to reference these structures

Maps defined to Mapping Services are loaded automatically the first time they are referenced from NCL. They can also be loaded manually by entering the LOAD MAP=mapname command.

The LOAD MAP command will only load a map not already in memory. To reset a map in memory (for example, after modifying an existing map), use the UNLOAD MAP=mapname command. It will be automatically reloaded on demand.

Connection to Mapping Support

An MDO is usually attached to mapping support at the same time it is created or data is first placed into it. The MAP= operand is available on many verbs and is used to attach a map to an MDO.

Example: Connect to Mapping Support

```
&ASSIGN MDO=xxx MAP=MYMAP
                -* This creates the MDO if it didn't already
                -* exist, and assigns a map to it.
&FILE GET ID=FILEID MDO=yyy MAP=MYMAP2
                 -* This creates the MDO if it didn't
                 -* exist, places data from the file record into
                 -* it, and attaches the map to it.
&APPC SEND...  VARS=a* MAP=MYMAP2
                 -* This transmits the data in the variables, and
                 -* the mapname.
&APPC RECEIVE...  MDO=zzz
                 -* This receives the data into the MDO, and
                 -* automatically attaches any map received to the
                 -* MDO as well.
```

Sourcing Data

MDO data can be sourced from NCL tokens, many NCL input verbs or as a hard-coded string. An MDO does not have to be attached to a map before data can be placed into it, because MDOs and maps are separate entities. It is possible to attach a map to an MDO before or after data has been placed into it.

Example: Sourcing Data

```
&ASSIGN MDO=ccc DATA=data
                -* assign data into the MDO
&ASSIGN MDO=ccc MAP=mapname
                -* attach a map to the MDO.
```

Note: After map connection, the &ZMDORC system variable should be checked to ensure the connection was good. If Mapping Services detects a mismatch between the map definition and the MDO data, &ZMDORC contains a non-zero value and subsequent access to the MDO can produce name checks or type checks.

Manipulate and Extract Data

Data in an MDO is manipulated using the &ASSIGN statement. There are two major operand forms on the &ASSIGN statement; an MDO stem name, and a compound name.

Stem Name: MDO=vvv

Compound Name: MDO=aaa.bbb.ccc

The stem name is used to refer to the entire data portion of an MDO. An MDO does not have to have a map attached to it to be referenced by its stem.

The compound name is used to refer to a structure within the MDO. A map must be attached to the MDO for this type of reference, otherwise NCL will not be able to locate the structure. (NCL cannot make sense of the data without a map.)

The &ASSIGN verb is useful when extracting individual components in an MDO into NCL tokens.

Use a Map in NCL Processing

After a map is defined, it can be used to interpret the contents of MDOs in NCL.

The following example demonstrates how MDOs can be used for the encapsulation and transmission of data in NCL.

Procedure A can receive messages from procedure B, and perform some desired action on some of the data in the message. If the action is completed successfully, procedure A will return the modified data to the initial sender (procedure B), with a message indicating the operation was successful. If the action is not completed successfully, procedure A will carry out some extra error processing and then return a message to the sender indicating that the operation was unsuccessful.

The message data is mapped with a map that contains the following components:

ACTION

The action to be performed.

ASN.1 Type: GraphicString

DATA

The data on which to perform the action.

ASN.1 Type: OCTET STRING

ACTIONRESULT

The result of the action processing.

ASN.1 Type: ENUMERATED

Procedure A could be as follows:

```
.RECEIVELOOP
   &INTREAD MDO=ACTIONPARMS  -* Read the MDO from the sender
   &ASSIGN VARS=SENDER FROM MDO=$INT.SOURCE.NCLID
                             -* Extract the senders NCLID
                             -* from the system MDO, $INT,
                             -* which is always set after an
                             -* &INTREAD operation, and is
                             -* mapped by the $MSG system map
   &ASSIGN VARS=ACTION FROM MDO=ACTIONPARMS.ACTION
                             -* Extract the type of
                             -* action to be performed
   &ASSIGN VARS=ACTIONDATA FROM MDO=ACTIONPARMS.DATA
                             -* Extract the data on which
                             -* to perform the action
-EXEC ACTPROC ACTION=&ACTION ACTIONDATA=&ACTIONDATA
                             -* Call proc to carry out
                             -* action
   &IF &RETCODE EQ 0 &THEN + -* Check return code from action
      &DO
         &ASSIGN MDO=ACTIONPARMS.DATA FROM VARS=ACTIONDATA
                             -* Set modified data in MDO
         &ASSIGN MDO=ACTIONPARMS.ACTIONRESULT DATA=OK
                             -* Set result of action in MDO
         &WRITE NCLID=&SENDER MDO=ACTIONPARMS
                             -* Return modified MDO to caller
      &DOEND
   &ELSE +
      &DO
         &GOSUB .ERRORPROC    -* Do error processing
         &ASSIGN MDO=ACTIONPARMS.ACTIONRESULT DATA=FAIL
                             -* Set results of action in MDO
         &WRITE NCLID=&SENDER MDO=ACTIONPARMS
                             -* Return MDO to caller
      &DOEND
&GOTO .RECEIVELOOP           -* Loop to process next message
```

You can see that Mapping Services greatly simplifies the NCL required to process complicated data formats. Often, externally-sourced data has a format which is awkward to manage in NCL. In these situations, Mapping Services can prove to be a very useful tool.

Chapter 14: Using Mapping Services

This section contains the following topics:

Overview

Mapping Services uses the ISO standard Abstract Syntax Notation One, or ASN.1, as the map definition language. The added sophistication that ASN.1 brings to the definition phase leads to greatly improved application use of Mapped Data Objects (MDOs).

By understanding ASN.1 and how Mapping Services implements its concepts, significant advantages in NCL processing of complex data structures can be realized. Using ASN.1 in defining maps, all MDO components have a data type. Only data that is of the correct type for the component can be assigned into a component. An attempt to place an invalid value into an MDO component fails with a type check return code.

Two types of data are available:

- Simple—for simple data types, the assigned data must be of the correct format for the component type. Otherwise, type check results.
- Constructed—for constructed data types, entire construction must be valid according to the logical type (described by the ASN.1 type definition) and the physical type (described by any Mapping Services implementation-specific definitions). Otherwise, type check or data check results.

MDO Behavior and NCL Processing Conventions

After using any verb that references an MDO, the MDO return code (&ZMDORC) and feedback (&ZMDOFDBK) system variables are set. Therefore, check most verbs when using operations involving MDOs.

Because &ASSIGN is used more frequently to process MDO data, an alternative exists. If &CONTROL MDOCHK is in effect, any error situations that normally result in a return code of 8 or higher cause the NCL procedure to terminate, but only if &ASSIGN sets the return code or feedback variable.

If &FILE GET MDO=*xxx* sets &ZMDORC to greater than 8, the process will not terminate if &CONTROL MDOCHK is in effect. However, if the default &CONTROL NOMDOCHK is in effect, all error checks are reported through the return code and feedback values.

The possible values of the return code and feedback system variables and their meanings are shown in the following table.

&ZMDORC	&ZMDOFDBK	Meaning
0	0	ok
4	0	null: optional component present but empty, or null data assigned to optional component
	1	null: optional component not present
	2	null: mandatory component present but empty, or null data assigned to mandatory component
	3	null: mandatory component not present
	4	string was truncated (applies to FIX offset or length components only)
8	0	type check: data is invalid for type
	1	data check: data is invalid structurally -a common cause is data too long or too short
12	0	name check: component not defined
	1	name check: index position invalid or value is out of range

&ZMDORC	&ZMDOFDBK	Meaning
16	0	map check: map not found
	1	map check: map contains errors, load failed
	2	map check: map/data mismatch

To minimize the use of these return codes it is necessary to understand the general behavior of MDOs with NCL.

An &ZMDORC value of 0 means that the data referenced was of a valid type for the component referenced. On assignment, the data is formatted and placed in the target component, resulting in an &ZMDOFDBK value of 0.

An &ZMDORC value of 4 is returned when an MDO component has a null, or empty, value (unless it was the NULL type in which case &ZMDORC is 0). Any component, regardless of its type, can be set to a null value. &ZMDORC is also set to 4 when a string type is truncated and an &ZMDOFDBK value of 4 is returned, but only if it is a fixed-length component.

An &ZMDORC value of 8 is returned if &CONTROL NOMDOCHK is in effect, and the data referenced does not conform to the type of the component referenced or cannot be assigned for other reasons. In exceptional circumstances, a type check or data check can result on a read intent operation. More usually, however, it occurs on an update intent operation where the data being assigned is invalid for the data type and a type check results. If on an update operation the function cannot be performed for other reasons, due to insufficient available space in a component that cannot be further extended, a data check results. In all such cases the operation fails, and the referenced MDO component is unchanged.

An &ZMDORC value of 12 is returned under the following conditions:

- &CONTROL NOMDOCHK is in effect and the MDO is mapped but the specific component referenced was not defined in the map
- The name is valid, but an index was used on a component that is not allowed to be indexed
- The index exceeds the defined index range

An &ZMDORC value of 16 is returned if, on any verb, a map connection request fails. &ZMDOFDBK indicates the reason that the map connection failed.

Other system variables available to interrogate error conditions are:

&ZMDOID

Contains the identifier of the last known MDO involved in the last operation.

&ZMDOMAP

Contains the map name for &ZMDOID.

&ZMDONAME

Contains the fully qualified name of the MDO component involved in the last operation.

&ZMDOCOMP

Contains the name of the component. The value is the last name segment of the fully qualified name for the MDO component involved in the last operation, if applicable.

&ZMDOTYPE

Contains the type of &ZMDOCOMP, if applicable.

&ZMDOTAG

Contains the tag value of the component involved in the last operation, if applicable.

Input Operations on an MDO

A number of NCL verbs allow input operations on an entire MDO. These are:

- &APPC RECEIVE
- &ASSIGN
- &CNMREAD
- &DECODE
- &ENCODE
- &FILE GET
- &INTREAD
- &LOGREAD
- &MSGREAD
- &PPI RECEIVE
- &PPOREAD
- &VARTABLE GET

When the MDO is targeted for input the MAP operand is allowed to define the mapping of the data object being accessed. The state of the MDO following any such input operation is determined by a number of factors that apply generally to all verbs. Together these considerations produce an MDO behavior which is predictable, as follows.

- If the verb return code indicates that the request was not satisfied, either due to some error, or because no data satisfied the particular request (perhaps due to timeout, or selection criteria or similar), the target MDO is deleted. Subsequent reference to the MDO or its components is invalid, and will produce a name check return code.
- If the verb return code indicates that the request was satisfied, then the target MDO always exists, even if it is null (or empty). If no map name was supplied on the input operation, and no default map name applies, then the MDO is unmapped. If a map name was supplied but either the map could not be found, was in error, or the data did not conform to the map definition, then the map check return code is set with a feedback indicating the nature of the error. If the map was not found, the data is present in the MDO which is unmapped. Otherwise, if no errors are encountered, the MDO will exist and is mapped according to the map name implied.

After performing a successful input operation on an MDO, the &ZMDORC system variable should always be checked to ensure that the outcome was good, and that the MDO is still mapped. Failure to do so can cause the NCL procedure to be terminated if a reference to an MDO component is made, the MDO is unmapped, and &CONTROL MDOCHK is in effect.

However, once the MDO is bound to the map without error, its contents are guaranteed valid by Mapping Services and there is usually no need to check the return codes for every access to MDO components, but they are available if required.

Output Operations from an MDO

A number of NCL verbs allow output operations from an entire MDO. These include the following:

- &APPC SEND
- &ASSIGN
- &CNMALERT
- &CNMSEND
- &DECODE
- &ENCODE
- &EVENT
- &FILE PUT
- &PPI SEND
- &VARTABLE PUT
- &WRITE

Read exceptions from an MDO are rare, and are confined to a name check if an undefined component is referenced, or a data check if the data does not conform to the mapping rules. However, in some instances where the component definitions of fixed fields overlap each other type checks on read are possible.

&ASSIGN Verb

The &ASSIGN verb provides the only access to and from individual components within an MDO.

Note: For more information about the &ASSIGN verb, see the *Network Control Language Reference Guide*.

Create and Delete MDOs

To create a mapped MDO using the &ASSIGN verb, issue *one* of the following statements:

```
&ASSIGN MDO=mdo MAP=mapname [ DATA=data ]
```

```
&ASSIGN MDO=mdo MAP=mapname [ FROM VARS=vars...  ]
```

To copy an MDO:

```
&ASSIGN MDO=mdo FROM MDO=sourcemdo
```

To create an unmapped MDO:

```
&ASSIGN MDO=mdo DATA=data
```

To delete an MDO entirely:

```
&ASSIGN MDO=mdo
```

Assignment of Data into an MDO

The &ASSIGN OPT=DATA option is the default and can be used to set MDO components from:

- Another MDO component
- One or more NCL variables
- User-supplied data

When assigning into an MDO component of a simple type from NCL variables or constant data, the input supplied must be in the valid external form for the component type or a type check results.

When assigning into an MDO component of a simple type from another MDO component the input selected must have a local form valid for the target component type. If the types are different, type conversion takes place where possible. Otherwise, a type check results.

When assigning into an MDO component that is a constructed type, the following conditions must be satisfied:

- The data must be valid in its physical format according to the structuring rules for the target component.
- Each embedded component must be in its valid local form.

If the conditions are not satisfied, a type check results.

Note: For more information about the external form and local form of data for each type, see *Managed Object Development Services Guide*.

Assign into/from a Single MDO Component

The following assign statements can be used to set the value of a single target MDO component, which can be the entire MDO, from NCL variables or constant data:

`&ASSIGN MDO=`*`a.b.c`*` DATA=`*`xxx`*

`&ASSIGN MDO=`*`a.b.c`*` FROM VARS=`*`vars`*`...`

In either case the input string must be valid external form for the component or a type check results.

When multiple NCL variables are specified as the source data, they are concatenated together to form the input string. The exception to this is if the assignment is into an entire MDO mapped by $NCL. In this case, a standard variable structure is built and maintains the variable boundaries.

The following assign statement can be used to get the value of a single target MDO component, which can be the entire MDO, into NCL variables:

`&ASSIGN VARS=`*`vars`*` FROM MDO=`*`a.b.c`*

In all cases, the result is a valid external form for the component unless a type check or data check occurs.

When multiple NCL variables are specified as the target data, they are segmented according to the maximum variable size (or specific segment sizes if supplied) from the entire output string. The exception to this is if the assignment is from an entire MDO mapped by $NCL. In this case, the NCL variables are updated according to the variable boundaries within the MDO.

The following assign statement can be used to move the value of a single target MDO component to another MDO component:

`&ASSIGN MDO=`*`a.b.c`*` FROM MDO=`*`x.y.z`*

In this case the assignment takes place using the normal local form for the component (not the external form) and as usual unless the input is valid, a type check results.

Assign into/from Multiple MDO Components within a SEQUENCE or SET Type

When an MDO component is a structure defined with a type of SEQUENCE or SET, you can assign into or from some or all of the components that comprise the structure by name. This is a generic form of assign with the following possible options and syntax:

```
&ASSIGN MDO=a.b.*
      { GENERIC | ADD | REPLACE | UPDATE }
        DATA=data
```

```
&ASSIGN MDO=a.b.*
        FROM
        VARS=vars*
      { GENERIC | ADD | REPLACE | UPDATE }
```

```
&ASSIGN MDO=a.b.*
      { GENERIC | ADD | REPLACE | UPDATE }
      { FROM | PRESENT_IN | DEFINED_IN }
        MDO=x.y.*
```

```
&ASSIGN VARS=vars*
      { GENERIC | ADD | REPLACE | UPDATE }
      { FROM | PRESENT_IN | DEFINED_IN }
        MDO=x.y.*
```

Note: The asterisk (*) must be in place of the last component name only.

In all cases:

- The multiple assignment proceeds as though a separate assignment was issued for each component selected.
- The PRESENT_IN and DEFINED_IN keywords apply only to a source MDO, not a target MDO.
- The GENERIC, ADD, REPLACE, and UPDATE keywords affect target MDO components or NCL variables only.

For all options, when assigning data into an MDO, the process is driven by the components defined within the map for the target MDO. Each component defined within the parent structure is a target for an assign.

- If the DATA keyword is used as the source, all target components are subject to assignment of the same data value.
- If the VARS keyword is used as the source, each target component name is used to access a source NCL variable. The name of the variable is constructed by appending the component name to the supplied VARS prefix.
- If the MDO keyword is used as the source, each target component name is used to access a source component. The name of the source component is constructed by adding the component name as the last name segment in the generic source MDO name.

When selecting data for assignment from an MDO, the FROM keyword is used to select only source components that have a data value that is not null. The PRESENT_IN keyword is used to select from all source components that are present. For both FROM and PRESENT_IN options, any component that is defined but not present is deemed to be null. When the DDEFINED_IN keyword is used, all components defined in the target map for the generic name level indicated take part in the operation, regardless of whether any data exists. Components not defined are deemed to be null.

When assigning data from an MDO into NCL variables, the process is driven by the components defined in the source MDO. Target NCL variable names are constructed by appending selected component names to the supplied VARS prefix.

When the GENERIC keyword is used, any existing target NCL variables or MDO components are first deleted, then each is assigned the value from the corresponding source component. If no source data exists no assignment takes place.

When the ADD keyword is used, only those NCL variables not currently present, or MDO components defined but not currently present in the target structure, take part in the assignment process. That is, only new NCL variables or MDO components are added and no existing ones are affected.

When the REPLACE keyword is used, only those NCL variables that are present, or MDO components that are defined and are present in the target structure, take part in the assignment process. That is, no addition takes place. Only existing NCL variables that have new source data or existing MDO components that have new source data are affected, but those variables that have no new source data are not affected.

When the UPDATE keyword is used, both addition and replacement take place, but existing NCL variables or MDO components that have no new source data are unaffected.

Assign into/from Components within a SEQUENCE OF or SET OF Type

When an MDO component is a structure defined with a type of SEQUENCE OF or SET OF, you can assign into or from some or all of the components that comprise the structure by a generic index value. This form of assign uses a varying range, and the syntax is as follows:

```
&ASSIGN { MDO=a.b.{*} | VARS=aaa*}
          RANGE=(n,m)
        { DATA=data |
          { FROM { VARS=bbb* | MDO=x.y.{*} }
            [ RANGE=(p,q) ] } }
```

Note: The asterisk in braces ({*}) must replace a SET OF or SEQUENCE OF index only. The asterisk can appear once only anywhere within the MDO name referenced.

The target component names of the form a.b.{*i*}, where *i* = *n* up to *m*, take part in the assignment from the corresponding source variable. The multiple assignments take place as though a separate assignment was issued for each item within the SET OF or SEQUENCE OF structure. The variable index can be the last part of the MDO name (as shown in the previous example), or more name segments can follow (for example, MDO=*a*.*b*.{*}.*c*).

Query MDO Components

Once an MDO is connected to a map, it is possible to query its structure (as present in the MDO), or its definition (as defined in the map). The syntax used is part of an &ASSIGN, where the results of the query must be placed into NCL variables.

```
&ASSIGN OPT={ NAMES |
              TAGS |
              TYPE |
              LENGTH |
              #ITEMS |
              NAMEDVALUES |
              VALIDVALUES }
        VARS=vars...
      { PRESENT_IN | DEFINED_IN | MANDATORY }
        MDO=mdo_name
```

When the PRESENT_IN keyword is specified the information is returned only for those components that are found to be present within the MDO. When the DEFINED_IN keyword is used, the information is returned for all those components defined within the connected map, regardless of their presence or absence in the MDO itself. If the MANDATORY keyword is used then only those defined components that are mandatory are selected.

The various options and their meanings are as follows:

OPT=NAMES (or OPT=NAME)

Applies to PRESENT_IN, DEFINED_IN, and MANDATORY options and returns the component names associated with the target *mdo_name* as follows:

- If only an MDO stem name is specified (for example, MDO=*abc*), then the name of the connected map is returned as the defined component name, but only if the MDO exists.
- If *mdo_name* is a compound name (for example, MDO=*a*.*b*.*c*), the name of the last component in the name list is returned (that is, *c*), depending upon the PRESENT_IN, DEFINED_IN, or MANDATORY option.
- If the *mdo_name* is a compound generic name (for example, MDO=*a*.*b*.*c*.*), multiple names can be returned, where each name returned is a subcomponent of the nominated component. For example, for MDO=*a*.*b*.*c*.*, all components defined within "*c*" are returned. This format is useful in determining the names of all components that are either present in, or defined within, a given structure. It is also useful in determining which component is within a structure that is a CHOICE type. However, for SEQUENCE OF and SET OF items, it is possible to have null named components because the SEQUENCE or SET OF items are processed by index value only.

  A compound variable indexed name (for example, MDO=*a*.*b*{*}.*c*, or MDO=*a*.*b*.{*}) is not supported on this query.

OPT=TAGS

Applies to PRESENT_IN, DEFINED_IN, and MANDATORY options and returns the component tags used by Mapping Services associated with the target *mdo_name*. Component selection is as for OPT=NAMES.

OPT=TYPE

Applies to PRESENT_IN, DEFINED_IN, and MANDATORY options and returns the component type defined within the map and associated with the target *mdo_name*. Component selection is as for OPT=NAMES.

OPT=LENGTH

Applies only when PRESENT_IN is specified and returns the local form data length within the MDO of the target components. Component selection is as for OPT=NAMES.

OPT=#ITEMS

Applies only when PRESENT_IN is specified, and returns the number of items within a nominated component as follows:

- If *mdo_name* is a stem name (for example, MDO=*stem*) or a compound name (for example, MDO=*a.b.c*), then a count of 0 is returned if the component does not exist. Otherwise, it is 1.
- If *mdo_name* is a compound generic name (for example, MDO=*a.b.c*.*), a count of 0 is returned if the nominated component *a.b.c* is one of the following:
  - Does not exist
  - Exists but is empty
  - Exists but is not constructed

  Otherwise, it provides the number of components present within the structure *a.b.c*.
- If *mdo_name* is a compound variable indexed name (for example, MDO=*a.b*{*} or MDO=*a.b*.{*}), the number of components present in the SET OF or SEQUENCE OF structure is returned. If the structure does not exist or is empty, 0 is returned. The variable index must be in the last name segment.

OPT=NAMEDVALUES

Applies to components that have named values associated with their type. These types are limited to BIT STRING, INTEGER, and ENUMERATED. Other types return null results. No generic indexes or generic names are allowed on this option.

If DEFINED_IN is specified a list of the named values defined in the map for the specified component is returned.

PRESENT_IN is invalid for OPT=NAMEDVALUES.

OPT=VALIDVALUES

Applies to string types that can have their character set constrained to particular characters or strings. This option only works with the defined keyword. If a string type (for example, GraphicString) has been constrained to a particular set of characters or strings, then this option returns the valid characters or strings in the target variable. If there are no constraints then no values are returned on assignment. The &ZVARCNT system variable is set to indicate the number of target variables set by the assignment.

Example 1: Query MDO Components

The following component is defined:

```
datax GraphicString ("ABCD" | "xyz" | "QQQ")
```

You use the following statement to query the component:

```
&ASSIGN VARS=X* OPT=VALIDVALUES DEFINED MDO=... datax
```

The following variables are returned:

```
&X1=ABCD
&X2=xyz
&X3=QQQ
```

Example 2: Query MDO Components

The following component is defined:

```
datax GraphicString (FROM ( "A"c | "C" | "Y"C | "X" ))
```

You use the following statement to query the component:

```
&ASSIGN VARS=X* OPT=VALIDVALUES DEFINED MDO=... datax
```

The following variables are returned:

```
&X1=A
&X2=C
&X3=Y
&X4=X
```

NCL Reference, Type Checking, and Data Behavior

When referencing an MDO in an NCL procedure, Mapping Services validates that the named component is defined (according to the name hierarchy supplied), and that the data within the component is valid, according to its underlying ASN.1 type. Each ASN.1 type can contain only certain valid values. Mapping Services checks the data value when retrieving data from, or assigning data into, an MDO. An operation attempting to retrieve or assign invalid data is rejected by Mapping Services with a feedback indicating type check.

In order to perform type checking Mapping Services first determines the base ASN.1 type of the component. Where a component is of a user defined type, the base ASN.1 type of the user defined type is inherited by the component. It is possible to have a number of levels of indirection between a user defined type and its base ASN.1 type.

The valid NCL values allowed for each of the base ASN.1 types is termed the external form. In addition to the set of valid values for each type, a specific component can be further constrained in what values are acceptable. Such constraints can be the result of either ASN.1 definitions or compiler directives. Finally, when data representing a valid NCL value is accepted for a component update, it is subject to a transformation from external form to local form, which is the MDO internal representation of data. This process can carry with it further constraints.

The valid external form values, and the behavior of data managed by Mapping Services, is described for each type in the following sections.

Notes:

- In the following descriptions, all string types that are defined as fixed (using the --<FIX(n)>-- directive) are subject to padding and truncation, without any indication in the return codes.
- Any types constrained by the SIZE parameter are not subject to padding or truncation. The data supplied must be within the SIZE constraints specified, or a type check results.

BOOLEAN Type

The BOOLEAN type is used to represent a value of true or false only.

External Form - Input

The local character strings TRUE and FALSE (not case sensitive) are accepted, while the digit 0 is interpreted as false, and the digit 1 is true.

External Form - Output

The digit 0 (false) or 1 (true) is always returned.

Local Form and Behavior

Internally, Mapping Services stores a value of X'00' for false, and X'01' for true (and accepts any value other than X'00' as true).

For an input operation, where the component is variable length, its length is always set to 1. Where the component length is fixed and is greater than 1, the value occupies the first byte only (that is, it is left-aligned) and the remainder of the component's data is set to zeros.

For an output operation, where the component is located and has a length greater than 1, only the first byte is inspected as the value.

INTEGER Type

The INTEGER type is used to contain any positive or negative whole numbers in the range -2,147,483,648 to 2,147,483,647 (that is, it is a signed 32 bit number).

External Form - Input

Valid input consists of a string of up to 15 digits optionally preceded by a plus sign (+) or minus sign (-) providing the sign (positive or negative) of the value (positive if omitted). All other characters must be valid digits (that is, 0 through 9). Alternatively, if the map definition included named values for this component, the symbolic name of the named value can be supplied as external form input.

External Form - Output

Output consists of a string of one or more local characters. If the integer value is negative the first character is a minus sign (-), otherwise the sign is omitted. All other characters are numeric characters. Leading zeroes are stripped.

Local Form and Behavior

Internally, Mapping Services can store integers in one of three formats:

binary

Can be up to 4 bytes in length. If the length is not fixed then the value is kept in the smallest number of bytes possible. If the length is fixed then the value is right-aligned and sign extended to the left.

packed

Can be up to 8 bytes in length. The integer value is converted to the packed decimal equivalent. If the length is not fixed then the value is kept in the smallest number of bytes possible. If the length is fixed then the value is right-aligned and zero padded to the left.

zoned

Can be up to 15 bytes in length. The integer value is converted to the zoned decimal equivalent. If the length is not fixed then the value is kept in the smallest number of bytes possible. If the length is fixed then the value is right aligned and zero padded to the left.

For any format, if a value exceeds that which can be stored without loss of significance a type check results. If a named value is input then the map definition is used to determine the actual integer value.

BIT STRING Type

The BIT STRING type is used to contain any data where individual bit values might have meaning. Mapping Services supports two types of BIT STRING access, standard and boolean.

Standard BIT STRING Access

Standard BIT STRING access deals with the string as a whole, allowing manipulation of the entire component through a single operation, as for most other types.

External Form - Input

Valid external form can be a string of one or more digits, each a 0 or 1. However, where named values are defined for the BIT STRING type, a list of named values, each separated by a plus sign (+) or a minus sign (-), is an acceptable alternative. A named value preceded by a plus sign indicates that the named bit value should be set to true (the bit is set to 1), and a minus sign indicates that the value should be set to false (the bit is set to 0).

External Form - Output

The output format depends upon whether or not named values are defined for the BIT STRING type. Where no named values are defined the output consists of a string of zero or more (always a multiple of 8) digits, each a 0 or 1. Where one or more named values do exist the output is a character string comprised of every named value corresponding to a set bit preceded by a plus sign meaning the value is true (the named bit is 1). The names of bits which are not set are not returned.

Local Form and Behavior

When a string if 0's and 1's are supplied as input, each digit in the input sequence is treated (left to right) as the value of the corresponding bit in that position of the local form data. If the number of bits supplied is not a multiple of 8, then trailing bits are set to zero and padded to a byte boundary. If the component has a fixed length exceeding that of the input string the value is left-aligned, and all unreferenced bytes are set to X'00'. If the component cannot contain the number of input bytes supplied, a data check results.

When a list of named values, each preceded by a plus sign or a minus sign, is supplied as input only the named bits take part in the operation. Each named bit preceded by a plus sign is set to 1 (true), while each bit preceded by a minus sign is set to 0 (false). All other bits in the BIT STRING are unaffected by the input operation.

When fetching the value of a BIT STRING, a named value list is returned if any named values are defined for the type, else a string of 0's and 1's is returned corresponding to the BIT STRING values. When named values are defined, all other bits in the BIT STRING are ignored on output regardless of their value.

Boolean BIT STRING Access

Boolean BIT STRING access deals with individual bit level access and operates only through named values. This access is recommended because program access to bits is only via their symbolic named values, thus removing from NCL the need to know relative bit positions.

For Boolean BIT STRING access to be invoked, the named value of a bit is provided by NCL as an additional name segment after the BIT STRING component name. Since the BIT STRING type is primitive, the additional name in the name hierarchy is understood to be a named value, and is treated as a BOOLEAN type. No matter where the named value is in the BIT STRING the value of the bit is always 0 or 1, as for a BOOLEAN type.

External Form - Input

The local character strings TRUE and FALSE (not case sensitive) are accepted, while the digit 0 is interpreted as false, and the digit 1 is true. Null is a type check in this case.

External Form - Output

The digit 0 (false) or 1 (true) is always returned.

Local Form and Behavior

The component name plus the named value is treated as a reference to a specific bit (the bit position within the component being defined by the named value), and that bit is set to 0 or 1 depending upon the input. No other bits in the BIT STRING component are affected. If the component is extended to accommodate the input then all other bits are set to 0.

OCTET STRING Type

The OCTET STRING type is used to contain any data where no formatting is required.

External Form - Input

Any data.

External Form - Output

Data is returned unchanged.

Local Form and Behavior

Data is stored as is. If the component has a fixed length exceeding that of the input string, the data is left-aligned, and all unreferenced bytes are set to X'00'. If the component cannot contain the number of input bytes supplied, a data check results.

HEX STRING Type

The HEX STRING type is a Mapping Services extension to ASN.1, but is processed as a base ASN.1 type. It is identical in all respects to the ASN.1 OCTET STRING type except for its external form representation.

External Form - Input

Valid input consists of a string of one or more local characters, each selected from the set 0123456789ABCDEF. Each pair of hexadecimal characters represents a single byte value. If an odd number of characters is supplied the string is treated as though padded on the left with a single zero.

External Form - Output

Data is returned in hexadecimal characters, as for input. An even number of characters is always returned.

Local Form and Behavior

Each two hexadecimal characters of input represents the actual data to be stored in a single byte. Otherwise behavior is as for OCTET STRING.

NULL Type

The NULL type is used where data in a component either must be null (that is, empty), or not accessible.

External Form - Input

The only valid input is a null value.

External Form - Output

A null value is always returned.

Local Form and Behavior

The component can be created by an input operation, but no contents are modified. If it already exists no data is modified.

OBJECT IDENTIFIER Type

The OBJECT IDENTIFIER type is used to contain object identifier values that uniquely identify registered objects.

External Form - Input

Any sequence of characters (from the set 0123456789.) that does not begin or end with a period (.), contains no consecutive periods, but contains at least one period. Each sequence of decimal digits punctuated by a period represents a sub-identifier in the series of sub-identifiers that comprise an object identifier value.

External Form - Output

As for input.

Local Form and Behavior

The data format is as supplied for input, however truncation is not allowed. If the component is fixed length then it must be able to contain the input string, and if necessary will be padded with blanks, otherwise a data check results.

ObjectDescriptor Type

The ObjectDescriptor type is used to contain object descriptions for registered objects.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form and Behavior

The data format is as supplied for input. Normal string padding and truncation rules apply.

REAL Type

The REAL type is used to contain floating-point, or scientific notation, numbers in the range 10^{-70} to 10^{70}.

External Form - Input

+ or -

(Optional) Plus or minus sign.

nnnnnn

(Optional) Any number of digits.

period (.)

(Optional) Decimal place.

mmmmmm

(Optional) Any number of digits.

E*sxx*

(Optional) Signed exponent power of 10.

Range: -99 to 99

Either, or both, *nnnnnn* and *mmmmmm* are present, and the resulting REAL number is within the allowable range.

Examples: External Form - Input

```
14578923455096765442839404
-123.567
.555
.0023E-23
3.142776589E+66
```

External Form - Output

Normalized decimal real number.

+ or -

Plus or minus sign of the value.

.

Decimal place indicator.

nnnnnn

15 significant fraction digits.

Esxx

Signed exponent power of 10.

Examples: External Form - Output

```
+.314277658900000E-10
-.123456789000000E+52
```

Local Form and Behavior

For IBM mainframes, local form is a 64-bit long floating-point value, and the component must be at least 8 bytes in length. Truncation is not allowed, but if the component has a fixed length greater than 8 bytes the value is left-aligned and padded on the right with zero bytes.

ENUMERATED Type

The ENUMERATED type is used to constrain a component to a defined set of values. Each defined value is named using a name identifier similar to a component name. Associated with each name is a unique integer value (which can be signed), for example:

```
Color ::=ENUMERATED { red(0),blue(1),yellow(2),
                      green(3),black(7) }
```

External Form—Input

The external form must be one of the names listed in the ENUMERATED type. The enumerated values are not allowed (that is, red is valid, 0 is not).

External Form—Output

Same as input.

Local Form and Behavior

Internally, the ENUMERATED value is kept in the same manner, and is subject to the same local form constraints, as an INTEGER of the binary local form.

NumericString Type

The NumericString is a subset of GeneralString which comprises:

- 0 to 9 Numeric characters
- ( ) Space or blank character

External Form - Input

Any string of valid characters, as described.

External Form - Output

Same as input.

Local Form and Behavior

On input data is stored as supplied. Normal string padding and truncation rules apply.

PrintableString Type

The PrintableString is a subset of GeneralString which comprises:

- a to z (lowercase alphabetic characters)
- A to Z (uppercase alphabetic characters)
- 0 to 9 (numeric characters)
- ( ) (the space, or blank character)
- ' ( ) + , - . / : = ? (special characters)

External Form - Input

Any string of valid characters, as described.

External Form - Output

Same as input.

Local Form and Behavior

On input data is stored as supplied. Normal string padding and truncation rules apply.

TeletexString Type

Not used in NCL.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form and Behavior

As for OCTET STRING.

VideotexString Type

Not used in NCL.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form and Behavior

As for OCTET STRING.

IA5String Type

Transparent general character set. (VisibleString plus control characters).

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form and Behavior

On input data is stored as supplied. Normal string padding and truncation rules apply.

UTCTime Type

Date and time, as Universal Coordinated Time (year without century numbers), in the format:

YYMMDDHHMM[SS]Z

GMT date and time (to minutes or seconds); Z indicates GMT time

YYMMDDHHMM[SS]sHHMM

Local date and time (to minutes or seconds); with signed zone offset from GMT time (s = + or -)

External Form - Input

Any valid input, as described.

External Form - Output

Any valid data, as described.

Local Form and Behavior

On input data is stored as supplied. If the component has a fixed length then a short input string will be padded with blanks. Truncation is not allowed.

GeneralizedTime Type

Date and time, as GeneralizedTime (year includes century numbers), in the format:

```
YYYYMMDDHH[MM[SS]][.f]Z
```

GMT date and time (to hours, minutes or seconds); with optional fractional time units to any significance (hours, minutes or seconds); Z indicates GMT time

```
YYYYMMDDHH[MM[SS]][.f][sHHMM]
```

Local date and time (to hours, minutes or seconds); with optional fractional time units to any significance (hours, minutes or seconds); with signed zone offset from GMT time (s = + or -)

External Form - Input

Any valid input, as described.

External Form - Output

Any valid data, as described.

Local Form and Behavior

On input data is stored as supplied. Normal string padding and truncation rules apply.

GraphicString Type

Transparent character set of graphic characters only.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form and Behavior

On input data is stored as supplied. Normal string padding and truncation rules apply.

VisibleString Type

Transparent character set of graphic characters only.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form

On input data is stored as supplied. Normal string padding and truncation rules apply.

GeneralString Type

Transparent character set of both graphic and control characters.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form

On input data is stored as supplied. Normal string padding and truncation rules apply.

Type Conversion for MDO Assignment

Normally, when assigning data from one MDO component to another, the type of each component is identical so that data can be moved unchanged. However, if the target component is not of the same type as the source component type conversion is automatically performed where possible. Hence the result of the following assignment depends on the type of *a.b.c* and *x.y.z*:

```
&ASSIGN MDO=a.b.c FROM MDO=x.y.z
```

When assigning into an MDO from external data, or from NCL variables the data type of the input can be nominated for the MDO assignment. If the MDO component is not of this type then type conversion is performed, for example:

```
&ASSIGN MDO=a.b.c TYPE=INTEGER FROM DATA=123

&ASSIGN MDO=a.b.c TYPE=HEXSTRING FROM VARS=HEXDATA
```

Similarly, when assigning data from an MDO into NCL variables, the data type of the output can be nominated for the MDO assignment, for example:

```
&ASSIGN VARS=ABC FROM MDO=a.b.c TYPE=BITSTRING
```

Valid type operands are:

- BOOLEAN or BOOL
- INTEGER or INT
- BITSTRING or BIT
- OCTETSTRING or OCTET
- NULL
- OBJECTIDENTIFIER or OBJECTID or OID
- OBJECTDESCRIPTOR or OBJECTDESC or ODESC
- REAL
- HEXSTRING or HEX
- NUMERICSTRING or NUMERICSTR or NUMSTR
- PRINTABLESTRING or PRINTABLESTR or PRTSTR
- IA5STRING or IA5STR
- UTCTIME or UTC
- GENERALIZEDTIME or GTIME
- GRAPHICSTRING or GRAPHICSTR or GRAPHSTR

- VISIBLESTRING or VISIBLESTR or VISSTR
- GENERALSTRING or GENERALSTR or GSTR

Type conversion is attempted regardless of the source and target types. Thus, depending on the actual value of the source data and target type, some assignments produce valid results, while others produce a type check.

The ASN.1 types can be classified into three groups:

- Graphic-oriented types:

  OBJECT IDENTIFIER, OBJECT DESCRIPTOR, UTCTime, GeneralizedTime, NumericString, PrintableString, TelexString, VideotexString, IA5String,GraphicString, VisibleString, and GeneralString.
- Numeric-oriented types:

  BOOLEAN, INTEGER, BIT STRING, REAL, and ENUMERATED.
- Transparent types:

  OCTET STRING, HEX STRING, SEQUENCE (OF), SET (OF), and ANY.

The following table shows the general rule for type conversion. The table lists the formats of the source data value used for assignment into the target type.

SOURCE\\TARGET	**Graphic-oriented**	**Numeric-oriented**	**Transparent**
Graphic-oriented	External form	External form	Local form
Numeric-oriented	External form	Local form	Local form
Transparent	Local form	Local form	Local form

Note: For more information about the acceptable external and local form values for each type, see the *Network Control Language Reference Guide*.

Graphic-oriented Source Type

Where the target is one of the graphic- or numeric- oriented types, the external form output of the source data is assigned to the target as though it was external input. The following is an exception: Conversion between UTCTime and GeneralizedTime results in the insertion/stripping of the century value (insert 19 if yy greater than 50, otherwise insert 20).

Where the target is one of the transparent types, the local form of the source data is assigned to the target unchanged; that is, the local form of the source data becomes the local form of the target. However, the external outputs of the source and target can differ depending on their respective types. In either case, a type check results if the data is invalid for the source or target types.

Example: Graphic-oriented Source Type

```
&ASSIGN MDO=a.b.utc TYPE=GTIME DATA=19921012145000+1000
                    -* target type is UTCTime
&ASSIGN VARS=RESULT FROM MDO=a.b.utc
                    -* returns &RESULT = 921012145000+1000
&ASSIGN MDO=a.b.num DATA=0123
                     -* source type is NumericString
&ASSIGN VARS=RESULT FROM MDO=a.b.num TYPE=REAL
                    -* output type is REAL
                    -* returns &RESULT = +.123000000000000E+03
&ASSIGN MDO=a.b.graph DATA=ABCD
                    -* source type is GraphicString
&ASSIGN MDO=a.b.hex FROM MDO=a.b.graph
                    -* target type is HEX STRING
&ASSIGN VARS=RESULT FROM MDO=a.b.hex
                    -* returns &RESULT = C1C2C3C4
```

Numeric-oriented Source Types

Where the target is one of the numeric-oriented or transparent types, the local form of the source data is converted to the local form of the target. The source and target local form values are equal in most cases, although their external form output can differ depending on their respective types. The following, however are exceptions:

- If the source type is REAL and the target type is either BOOLEAN, INTEGER or ENUMERATED, then floating point integer value conversion (with rounding) is performed on the local form of the source data. Similarly, integer to floating point conversion is performed if the process is reversed.
- Conversion of local form source data to a BOOLEAN type target, results in X'00' (FALSE) if the source is non-zero. Otherwise, the target local form is converted to X'01' (TRUE).

Where the target is one of the graphic-oriented types, the external form output of the source data is assigned to the target as though it was external input.

In either case a type check results if the data is invalid for the source or target types.

Example: Numeric-oriented Source Types

```
&ASSIGN MDO=a.b.int TYPE=REAL DATA=99.99
                    -* target type is INTEGER
&ASSIGN VARS=RESULT FROM MDO=a.b.int
                    -* returns &RESULT = 100 (value rounded)
&ASSIGN MDO=a.b.bool DATA=TRUE
                    -* source type is BOOLEAN
&ASSIGN VARS=RESULT FROM MDO=a.b.bool TYPE=HEX
                    -* output type is HEX STRING
                    -* returns &RESULT = 01 (that is, TRUE)
&ASSIGN MDO=a.b.bit DATA=11101
                    -* source type is BIT STRING
&ASSIGN MDO=a.b.num FROM a.b.bit
                    -* target type is NumericString
&ASSIGN VARS=RESULT FROM MDO=a.b.num
                    -* returns &RESULT = 11101000
```

Transparent Source Types

Regardless of the target type, the local form of the source data is assigned to the target unchanged, that is, the local form of the source data becomes the local form of the target.

An exception is that the conversion of local form source data to a BOOLEAN typed target results in local form X'00' (FALSE) if the source is non-zero. Otherwise, the target local form is converted to X'01' (TRUE).

The external outputs of the source and target values can differ, depending on their respective types.

A type check results if the data is invalid for the source or target types.

Example:Transparent Source Types

```
&ASSIGN MDO=a.b.prt TYPE=OCTET DATA=AB.C
                   -* target type is PrintableString
&ASSIGN VARS=RESULT FROM MDO=a.b.prt
                   -* returns &RESULT = AB.C
&ASSIGN MDO=a.b.hex DATA=FF
                   -* source type is HEX STRING
&ASSIGN VARS=RESULT FROM MDO=a.b.hex TYPE=INT
                   -* target type is INTEGER
                   -* returns &RESULT = -1
&ASSIGN MDO=a.b.any DATA=DEFG
                   -* source type is ANY
&ASSIGN MDO=a.b.hex FROM MDO=a.b.any
                   -* target type is HEX STRING
&ASSIGN VARS=RESULT FROM MDO=a.b.hex
                   -* returns &RESULT = C4C5C6C7
```

Chapter 15: NDB Concepts

Note: Familiarity with standard UDBs is assumed, including the use of the &FILE verbs, and the UDBCTL command.

This section contains the following topics:

What Is an NDB?

NDBs let NCL programmers define, create, update, and search a database based on a collection of user-defined fields. NDBs support information storage and retrieval through the CA NetMaster and SOLVE functions. Your product provides this enhanced database manipulation facility for NCL procedures.

For example, a SHOW NDB=ALL command typically shows up to five NDBs, depending on the product.

This facility allows data to be stored, to be retrieved or updated later, in a formatted database known as a NDB. The NDB format is more powerful than standard VSAM data sets.

Note: In this guide, the term database manager refers to the assembler code that controls NDBs. There is a database manager for each active NDB.

An NDB is a formatted VSAM key-sequenced data set (KSDS). It should be accessed only by using the &NDB verbs. An NDB supports the following:

- Multiple keys, without any VSAM alternate indices
- Logical record size not limited by the defined VSAM record size
- Data access by named field, not relative field position in a record (as in a UDB)
- Transaction integrity, guaranteeing a non-corruptible file
- Multiple users, without VSAM string limitations
- Different data types, including character, numeric, floating point, hexadecimal, and date format data. The Database Manager prevents data that is not in the defined format and invalid data, from being stored (for example, the value ABC could not be stored in a field defined as numeric).
- An extremely powerful search capability, that makes full use of keys wherever possible, but does not require any keying of the search arguments.
- Null field support, which allows multiple record types to co-exist in a single NDB.
- Forward recovery facilities, minimizing the risk of data loss.

Work with NDBs

You can use NDB commands to start, stop, reset, and lock (for example to prevent access while running a backup), an NDB. For more information, see the Online Help.

You can also use NCL verbs to insert or delete field definitions, add, delete, update and retrieve records, and to search an NDB for all records that match a supplied set of criteria. These verbs also allow access to the database and field level definitions, making it easy to provide utility procedures that need only be told the name of the NDB to be accessed.

Uses of NDBs

An NDB can be used in any application where a flexible data storage mechanism is required. Applications that have complex retrieval needs are especially suited to NDBs. The ease of access of data makes such things as selection list scrolling very easy to perform.

NDBs are not suited to applications that have a large amount of high-speed record addition, update, or deletion.

Differences Between NDBs and UDBs

Both NDBs and UDBs are always VSAM data sets. An NDB is always a VSAM KSDS, whereas a UDB can be either a KSDS or an ESDS (a non-keyed VSAM data set).

The UDBCTL command is used to open a VSAM data set for access by the Virtual File Services (VFS). Thus, although an NDB should not be used as a UDB, the UDBCTL command is still used to open and close it.

Important! Do not access an NDB using the &FILE verbs, as if it were a UDB. If any access is done in this way, the NDB can be corrupted.

The following table summarizes the differences between UDBs and NDBS:

UDB	NDB
Can be a VSAM KSDS or ESDS.	Always a VSAM KSDS.
If KSDS, key length can be from 1 to 255.	Key length must be from 16 to 255.
VSAM maximum data length can be any valid value.	VSAM maximum data length has a minimum value restriction
Data is built or accessed by relative position in a record.	Data is accessed by field name.

UDB	NDB
Record length is limited by either the VSAM record length or the maximum NCL statement length after substitution.	Record length is independent of the VSAM record length or NCL statement length. The length is logically limited by the restriction of the 32 KB named fields.
Only one, unique, sequence key is provided, unless VSAM alternate indexes are used.	There is no requirement for any key. There can be any number of keys, and they need not be unique. There can, optionally, be a unique sequence key.
Multiple record structures to support multiple keying, and so on. Have no update integrity. A system failure can logically or physically corrupt the database.	Internal update journaling guarantees integrity of the multiple VSAM record structures used to support the NDB features.
Concurrent access can result in VSAM string waits or lockouts.	Concurrent access has no restrictions.
Complex searching is slow, as the logic must be implemented in NCL.	Complex searching runs at assembler speed. Keys are used whenever possible.
Multiple positioning (for sequential retrieval) by one NCL procedure is not possible.	An NCL procedure can have any number of simultaneous positions in an NDB.
UDBs are accessed using the &FILEOPEN verb. The first &FILE OPEN for a given UDB performs a logical open for the NCL process. Because the &FILE GET and similar verbs have no way of specifying the UDB they refer to, the last executed &FILE OPEN statement sets the UDB for these statements. Thus, &FILE OPEN has a double meaning.	NDBs are accessed using the &NDBOPEN verb. Each NDB accessed must be opened by an &NDBOPEN statement. The other &NDB*xxx* verbs allow specification of the database name and there is no need for repeated &FILE OPEN-like use of the &NDBOPEN verb.
Data in a UDB has no associated type. No validity checking is performed on the stored data.	Data in an NDB has a type. Invalid data (for example, nonnumeric for a numeric field) cannot be stored.

When an NDB is active, VFS prevents any access to it as a UDB. For example, if an &FILEID statement refers to an active NDB, it causes the NCL procedure to terminate with an error message.

Similarly, if an NDB is being accessed as a UDB, it cannot be accessed by NDB commands or &NDB statements.

NDB Types

The RAMDB (Automation Database File) and RSDB (Network Model File) are stable NDBs that rarely require attention; however, they should be backed up periodically to secure changes. For example, changes to resource definitions and Automation Services processes are stored in the RAMDB.

The Alert History File ($ALERTH), File Transfer Event Database (EVNTDB), and IPLOG Event History File (IPLOG) are used to store data cumulatively for searching, which is why they have user-defined parameters such as the number of days to keep data.

NDB Structure

The internal structure of an NDB is described in this section. The records are described in ascending VSAM key sequence.

Control Record

The first record (key is always all binary 0) in an NDB is a control record. It identifies this data set as an NDB, and contains other control information (for example, the number of defined fields and the number of records). This record is inserted when the NDB CREATE command formats an NDB. It is updated by other NDB commands and NCL statements. It also contains the domain ID (JCL parameter NMD ID) and NDB name. These are used to prevent concurrent update access by more than one region.

Journal Control Record

This record, also inserted when the NDB is formatted, is used to manage the NDB transaction journal (see page 293).

Journal Data Records

These records are used to journal update activity, to allow update retry after a system failure. There are n of these records, the number being determined either by the value of the NDBLOGSZ SYSPARM when the NDB CREATE command was issued for the NDB, or by the LOGSIZE parameter on the NDB CREATE command.

Field Definition Records

These records, one for each field defined on the database, contain information about the fields (for example, the field name, data format, and key options).

Key Statistics Records

These records, one for each field, contain key statistics information collected during an NDB ALTER BLDX or an NDB START KEYSTATS run.

Key Records

For fields defined as keyed (except a sequence key), these records contain the key values, and, for each key value, a list of the record IDs of records containing that value.

RID-Sequence Key Records

For NDBs defined with a sequence key, these records act as a link record, keyed by record ID, and contain the sequence key value of the record.

Data Records

These records hold the actual data for each record stored in the NDB. They are keyed by record ID, or by sequence key value (if a sequence key is defined for this NDB). If a given logical record has more data than will fit into a single VSAM record, it is automatically spanned across multiple records.

Record ID

Each record in an NDB is assigned a Record ID (RID), which is unique identifier of a logical record in an NDB.

Note: All dynamic data NDBs do not require reorganization to reclaim free space.

The RID is assigned by the Database Manager, when a record is added to an NDB. The RID is a number, from 1 to 1 billion (actually, 2**30 - 1), that uniquely identifies this record in the NDB. All internal access to a record is made by providing the RID. The RID assigned to a record never changes, and (currently) is not reassigned to another record when a record is deleted.

Note: The RID cannot be assigned by the user. The assigned value is made available to the NCL process that inserts a record via the &NDBRID system variable.

The RID is necessary, as an NDB does not need to have a sequence key (or any key, for that matter), and a way of uniquely identifying a record is always required.

Whenever a record is retrieved by an &NDBGET NCL statement, the RID of that record is returned in &NDBRID. The record could have been retrieved by a key, or from an &NDBSCAN result list, but the RID is always made available, so that a following &NDBUPD or &NDBDEL statement can refer to the correct record.

RID Reuse

RID reuse accumulates ranges of unused RIDs from deleted records during a KEYSTATS run, which is performed automatically during the daily old record deletion process for NDBs such as the Alert History File. By reusing RIDS, free space in the VSAM cluster is also reused.

Note: Do not reorganize NDBs when RID reuse is active.

You can find the messages in the log output from an $ALERTH daily purge by looking at the purge or delete time for the file in the Customizer Parameter group (/PARMS), and then locating that time in the log using the T hh.mm command.

Alerts for VSAM Monitoring

You can create alerts for VSAM files allocated to your CA Mainframe Network Management product region. This includes the NDBs because they use the same VSAM API. Event Distribution Services (EDS) creates events for file open, close, allocate, unallocated, and so on.

In addition, you can also use the following events that are triggered by VSAM return and feedback codes:

- $$SYS.FILE.EOV for End Of Volume (new data or index extent)
- $$SYS.FILE.CA.SPLIT for a VSAM control area split (not used by the VSAM monitor)
- $$SYS.FILE.FULL for a file full condition
- $$SYS.FILE.SHORTAGE for a string or buffer shortage
- $$SYS.FILE.ERROR for a VSAM error condition

These events are trapped by the VSAM monitor and alerts are raised if required. Log files and files opened for input are excluded from alerts related to file size because log files wrap and input files cannot be extended by the region's activity.

Alerts for String / Buffer shortage or file extensions are optional. If you do not want the IPLOG or EVNTDB to be automatically resized, you should clear the file extension severity count; otherwise, resize is attempted when the Extents count is exceeded.

NDB Data Formats

An NDB supports the following data formats:

CHAR

Data is provided, and stored, as a character string. NCL restricts character data to printable characters. The maximum length of a character field is 255 characters if not keyed, or (VSAM key length - 8) if keyed. Character fields collate (for keying, or sorting in a scan) on ascending EBCDIC value. Trailing blanks are not significant and are removed from stored data. The option to automatically make data upper case is available. Alternatively, data can be stored as lower case but searched as if it were upper case.

NUMERIC

Data is provided as an optionally signed number, from -2,147,483,648 to +2,147,483,647. Numeric fields collate on ascending binary value (-10 before -5 before 0 before +5 before +10). The minimum VSAM key length for an NDB guarantees that numeric data can always be keyed.

HEX

Data is provided as an even number of hexadecimal characters (0-9, A-F, or a-f). Trailing blanks are eliminated from the value. Trailing zeros are significant, and are stored. The maximum length of the character representation of a HEX field is 254, giving a maximum binary length of 127. If keyed, the maximum character representation length is ((VSAM key length - 9) * 2). A null-valued HEX field can be represented by a character value of one blank. HEX fields collate on ascending binary value, with values that are equal except for the number of trailing zeros collating on increasing length.

DATE

Data is provided in one of several formats, controlled by the user and/or system language code, and the current &NDBCTL DATEFMT setting. Basically, the provided value is in YYMMDD format. The data is stored internally as packed digits in the form YYMMDD. Date fields collate on ascending date value. The minimum VSAM key length for an NDB guarantees that date data can always be keyed.

As the DATE format does not include a century, CA recommends use of the CDATE format instead.

CDATE

Data is provided in one of several formats, controlled by the user and/or system language code, and the current &NDBCTL DATEFMT setting. The data is stored internally as a 3-byte binary number, being the numbers of days from 1/1/0001.

TIME

Data is provided in HHMMSS.TTTTTT format (the decimal point and fraction can be truncated or omitted). The data is stored internally as a 5-byte binary number, being the number of microseconds since midnight.

TIMESTAMP

Data is provided in YYYYYMMDDHHMMSS.TTTTTT format. The data is stored internally as a concatenation of a 3-byte CDATE and 5-byte TIME.

FLOAT

Data is provided as a floating point number. It is stored internally in IBM 8-byte normalized floating point format. The numbers are stored to 15 significant digits and with an exponent of +70. Floating point fields are collated on ascending numeric value.

Null Values and Null Fields

One of the most powerful features of NDBs is the use of null fields, an understanding of which is essential to the effective use of an NDB.

An NDB is a field-oriented database. Data is always accessed by field name. Other databases may have the concept of a record or group, being a collection of fields, that can be accessed by the name of the record or group. An NDB record is the actual, complete record, as logically accessed by RID.

At first, then, it might seem that an NDB can contain only one record-type, where record-type corresponds to a supplier, an order, or a customer record, for example. This is not the case. In fact, an NDB can contain almost any number of logical record types, each of which can be accessed separately. To achieve this, an NDB uses the concept of the null field. A null field is simply a defined field that is not present in a record.

The number of defined fields in the control record includes the number of null fields and each null field has a field definition record. However, the field is not physically present in the record. This is clearly not the same as a field that is present, but contains a null value (for example, blank).

Even if the field is defined as keyed, a null field is never keyed. Thus, any attempt to access by using keys will never retrieve a record that has that field null.

For any data format, a field can have one of the following logical values:

- Not present (that is, null field).
- Present, but null-valued. For a character field, this is defined as all blank. For a numeric or floating point field, this is defined as 0, for a hexadecimal field, this is defined as all blank (stored as present with a length of 0), and for a date field, this is defined as YYMMDD = 000000 (the only invalid date value allowed).
- Present, with a value other than the null value described previously.

In NCL terms, the following statements illustrate the three states of a field:

&VALUE =

Sets &VALUE to not-present (the actual variable is deleted from the NCL procedure's variable pool).

&VALUE = &SETBLNK 1

Sets &VALUE to present, with the value of one blank.

&VALUE = value

Sets &VALUE to present, with the value of value.

Rules for Null Fields

There are several rules regarding null fields:

- A null field is never keyed. Thus, access using keys for that field name will never retrieve a record with that field a null field.
- A null field never matches anything, not even another null field. For example, none of the following &NDBSCAN statements will retrieve the entire database, if field NAME is null in some records.

```
&NDBSCAN dbname     FIELD NAME EQ VALUE 'FRED' OR +

                    FIELD NAME NE VALUE 'FRED'
&NDBSCAN dbname     FIELD NAME EQ FIELD NAME
```

  In the second example, comparing a field to itself in a record, it might seem that some records should be selected. The rule about null fields is still honored in this case.

- You cannot retrieve a record with null fields. Retrieving a record with null fields causes the NCL variables receiving the requested null fields to be deleted (that is, set to null).
- A null field can be indicated on an &NDBADD statement by omitting the fieldname = fieldvalue clause for that field, or, on an &NDBADD or an &NDBUPD statement, by using a currently undefined (that is, null) NCL variable as the fieldvalue, for example, FIELDX = &NULL, or by using the syntax FIELDX NULL.
- The only way that records containing a particular null field can be selected in an &NDBSCAN is to use the PRESENT and ABSENT or IS [NOT] NULL operators. These operators allow records to be selected that contain the nominated field (PRESENT) or records that do not contain the nominated field (ABSENT).

Using null fields, the previous example of a database containing supplier, order, and customer records can be built by inserting only supplier fields for supplier records, order fields for order records, and customer fields for customer records. The records are now disjoint. A retrieval by CUSTNO, for example, would only retrieve records containing the CUSTNO field, and so on for supplier and order.

When defining fields in an NDB, a field can be made mandatory by specifying NULLFIELD=NO.

NDB Transaction Management: Database Protection

An NDB is protected against system failures that might occur when an update to the VSAM data set is in progress. Database record locking prevents data corruption caused by multiple users accessing the same record simultaneously.

If it is possible for more than one user to access an NDB record at once, use the &LOCK verb (see page 226) to ensure exclusive access to the record while it is being accessed. That is, perform an &LOCK on a record before any operation that accesses that record proceeds. If another user subsequently accesses the record, any modifications made by the second user do not proceed until the first accessing procedure concludes and the record ceases to be locked.

Protection against system failures is achieved as follows:

- A preformatted journal area is built when the NDB is created. This area consists of a journal control record, and n journal records.
- When an operation that involves updating the NDB starts, the journal area is used to record the updates, but they are not actually performed.
- When the updates are complete, the journaled updates are used to physically update the data set. Before this starts, a flag is set in the journal control record, indicating an update apply procedure is in progress, and the journal control record, and all journal data records are force-written to the VSAM data set.
- If the update apply completes successfully, then buffers are flushed, the journal control record flag is reset, and the journal control record is rewritten.
- If the update apply is interrupted by a system failure, then, when the NDB is next activated, the journal control record flag indicates that an update apply was in progress at the time of the failure. The entire update apply is redone (ignoring errors due to duplicate or already deleted records). Thus database integrity is assured.
- Physical errors on the VSAM data set (for example, out of space) are handled in the same way. Once the data set has been copied to a larger version, the reapply works in the same way.

NDB Journaling

In addition to the preformatted journal area kept within an NDB, an external journal can be created. This journal allows:

- Continuous availability of an NDB that cannot regularly be stopped for backups
- Recoverability of an NDB to the time of last update, even in the event of physical data set failure

To enable journaling on an NDB, specify the JOURNAL operand on the NDB START command. This operand causes the after images of all NDB record updates to be written to the journal.

Note: Before images are not kept; therefore data set back out is not possible.

Continuous Availability

If your NDB cannot be stopped for backup because of availability requirements, you can use the journal to keep a current backup copy. A backup copy is created once and the journal is applied to it each time the journal is swapped.

Note: A journal swaps if a JOURNAL SWAP command is issued, or if the current journal runs out of space.

Use the batch forward recovery utility (UTIL0010) to apply the journal. The name of the journal to be applied can be determined using the &ZJRNALT system variable. The sample batch forward recovery JCL ($NDUT010) is available in the distribution library and must be tailored using the installation data set names.

Whenever a journal swap occurs, an NCL procedure is started in the BSYS environment to assist in the automation of forward recovery. The name of this procedure is specified using the SYSPARMS JRNLPROC command (default is $NDJPROC). Use this procedure to submit your NDB forward recovery JCL.

NDB Recovery

If you intend to continue taking regular backups of your NDBs, then you can choose to apply journals only in the event of physical loss of an NDB. When taking regular backups of your NDB you only need to backup your journal data sets whenever a journal swap occurs. The NDB can then be restored from a backup, applying all journals since backup in sequence.

How to Respond to an Alert for File Size or File Full

When recovering a full or near-full NDB to a larger file, you must disconnect the old and new files completely from the region during the REPRO of old file records into a new, empty file so that there are no problems with control record corruption.

Note: The event data is not recorded during this time; however, the automatic resize of IPLOG and EVNTDB does cache data during the resize.

Fix a Corrupted NDB

An NDB full or region canceled event can corrupt an NDB. To recover the data, use the NDB ALTER OPT=BLDX command to rebuild the indexes.

Note: This procedure applies to an IPLOG file. Similar steps apply to all NDBs.

To fix a corrupted NDB

1. Create a work file. The size must relate to the number of records in the NDB.

   The following shows some sample JCL:

```
//DEFWRK  EXEC PGM=IDCAMS
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  *
 DELETE (hlq.TEST.NDBWORK) CL
 DEFINE CLUSTER (NAME(hlq.TEST.NDBWORK)      -
                 MGMTCLAS(DEFAULT)           -
                 STORCLAS(NMDPOOL)           -
                 INDEXED                     -
                 RECORDS(1000000 100000)     -
                 SPEED                       -
                 SHAREOPTIONS(2 3)           -
                )                            -
        DATA    (NAME(hlq.TEST.NDBWORK.D)    -
                 CONTROLINTERVALSIZE(4096)   -
                 RECORDSIZE(1017 4089)       -
                 FREESPACE(0 0)              -
                 KEYS(4 0)                   -
                )                            -
        INDEX   (NAME(hlq.TEST.NDBWORK.I)    -
                 CONTROLINTERVALSIZE(1024))
/*
```

2. Issue the following command from OCS to allocate the work file to the region:

   ```
   ALLOCATE DD=NDBWORK DSN=hlq.TEST.NDBWORK DISP=SHR
   ```

   Issue the following command to open the work file as a VSAM file.

   ```
   UDBCTL OPEN=NDBWORK STRNO=3 BUFNI=5 BUFND=6
   ```

3. Issue the following command to remove the NDB bad-locked status:

   ```
   NDB PURGE IPLOG
   ```

4. Issue the following command to stop the NDB to prevent access:

   ```
   NBD STOP IPLOG IMM
   ```

5. Issue the NDB ALTER command to verify the indexes, as follows:

   ```
   NDB ALTER IPLOG OPT=CHKX DB WORK=NDBWORK DETAIL=YES SORT=100000
   ```

   Minimizing I/O on the work file by specifying the maximum value for sort memory reduces execution time.

6. Issue the following command to close and free the work file:

   ```
   UDBCTL CLOSE=NDBWORK
   UNALLOC DD=NDBWORK
   ```

7. Review the CHKX. If it shows an error, rebuild the indexes by redefining the work file to empty it. Then, use the BLDX option of NDB ALTER, by issuing the following command:

   ```
   NDB ALTER IPLOG OPT=BLDX DB WORK=NDBWORK DETAIL=YES SORT=100000
   ```

   Minimizing I/O on the work file by specifying the maximum value for sort memory reduces execution time.

   If the NDB ALTER BLDX action fails, the work file must be unallocated and the file must be redefined before you reissue the command.

8. Reallocate the NDB by actioning the IPFILES - TCP/IP File Specifications Customizer Parameter Group.

Chapter 16: NetMaster Database Administration

Note: Familiarity with standard User Databases (UDBs) is assumed, including the use of the &FILExxx verbs, and the UDBCTL command. A knowledge of VSAM, the IDCAMS utility program, and JCL (for the relevant operating system) is also assumed.

This section contains the following topics:

How to Create an NDB

Perform the following tasks to create an NDB:

1. Define the VSAM data set (see page 298)
2. Calculate the key length (see page 298)
3. Calculate the record length (see page 299)
4. Allocate the VSAM data set (see page 300)
5. Open the VSAM data set (see page 300)
6. Create the NDB (see page 301)
7. Start the database (see page 302)
8. Insert field definitions (see page 302)
9. Load initial data (see page 303)

Read the complete description of each task carefully before performing the task.

Define VSAM Data Set

Use IDCAMS to define a VSAM KSDS. The following parameters are required:

INDEXED

Indicates a KSDS

KEYS (len 0)

key length (see Task 2)

RECORDSIZE (avg max)

Indicates average and maximum record size (see Task 3).

Other parameters, such as SPEED, REPLICATE, IMBED, can be used at your discretion.

Do not use SPANNED. Logical records longer than the VSAM data record length are handled automatically.

The REUSE parameter can be used, but, for NDBs used in a production environment, it is not recommended, as omission prevents the accidental emptying of an NDB by use of the UDBCTL OPEN RESET command.

Calculate Key Length

Calculate the key length to use for the NDB as follows:

L1 =

Longest desired length for any keyed CHAR field

L2 =

Longest desired length for any keyed HEX field (as displayable characters)

L3 =

Maximum of (L1 + 8), (L2 / 2 + 9), and 16

KL =

Minimum of L3, and 255.

Calculate Record Length

Calculate the average and maximum record lengths as follows:

Average Record Length:

- Greater than the VSAM key length
- The approximate size of a stored data record. A stored data record needs:
  - KL (from above) +10 +
  - 3 times number of present fields +
  - 4 times number of present numeric fields +
  - 3 times number of present date or cdate fields +
  - 8 times number of present floating point or timestamp fields +
  - 5 times number of present time fields +
  - Total number of characters in present hexadecimal fields / 2
  - Number of characters in present character fields (less trailing blanks) +

Note: A data record can span multiple VSAM records. If there are some records in your design that can be very large, but most are short, calculate the sizes using most common record structure.

Maximum Record Length:

- A maximum of 274 + 2 times the VSAM key length
- Average record length (calculated above) + VSAM key length + 6

VSAM requires the maximum record length to be less than the data CI size less 7. Use a figure that leaves minimum wastage in a CI.

The only VSAM records in an NDB that have the maximum VSAM record length are the NDB transaction data records. These are inserted during processing of the NDB CREATE command.

Example: VSAM Definition

This example of a definition is a good starting point for your own definitions.

```
DELETE clustername CL
DEFINE  CLUSTER(NAME(clustername)    -
        VOL(volume)                  -
        INDEXED-
        SHAREOPTION(2 3) -
        DATA (NAME(dataname)-
        KEYS(60 0)-
        CISZ(4096)-
        RECSZ(200 1020)-
        FSPC(20 20))-
        INDEX (NAME(indexname)-
        CISZ(2048))
```

Allocate the VSAM Data Set

Use the ALLOCATE command to dynamically allocate the data set to an active region:

```
ALLOC DSN=clustername DD=dbname DISP=OLD
```

Open the VSAM Data Set

Use the UDBCTL command to open and initialize the data set. Whether the database will use the VSAM LSR pool depends on the options specified on the UDBCTL command.

By convention, the database name is the same as the DD name (z/OS) or file name (z/VM).

If you do not want the database to use the LSR pool, issue the following UDBCTL command:

```
UDBCTL OPEN=dbname ID=* STRNO=7 BUFNI=10 BUFND=10
```

The values for STRNO, BUFNI, and BUFND are the suggested defaults.

The STRNO value should be 3 + (value of NDBSUBMX SYSPARM - 1, times 2). A lower value can lead to string space being dynamically acquired by VSAM (z/OS) or string waits (z/VM) if several &NDBSCAN statements are executing concurrently.

If you want the database to use the LSR pool, issue the following UDBCTL command:

```
UDBCTL OPEN=dbname ID=* LSR
```

Use the LSRPOOL command to define the LSR buffer pool sizes and the number of buffers for each size. LSR is the recommended way of running an NDB.

You might want to use deferred I/O when running the NDB. Deferred I/O involves sharing of buffers in between requests and enhances performance but possibly at the expense of integrity. Deferred I/O is not recommended when running an NDB in an online transaction update environment.

If you want to run deferred I/O you must use the DEFER option of the UDBCTL command to open the file:

```
UDBCTL OPEN=dbname ID=* LSR DEFER
```

You must also use the DEFER option on the NDB START command. For more information about the NDB START command, see the Online Help.

Create the NDB

To do this, issue the NDB CREATE command:

```
NDB CREATE dbname [LOGSIZE=n] [LOADMODE] [LANG=lc]
```

The NDB CREATE command formats the UDB into an NDB by inserting control records that identify it as an NDB, and builds journal records for transaction management.

The nominated data set (UDB, and so on) cannot be created into an NDB unless the data set is empty and it meets the requirements such as KSDS, key length, and record length.

The NDB can be created as language-specific by specifying the LANG= operand. The uppercase translation table for the language specified is then used for all uppercase processing.

Specify the number of log blocks to format with the LOGSIZE parameter. If you want to fast-load data, use the LOADMODE option to put the database into load mode.

Following the NDB CREATE command, the data set is now formatted as an NDB, with no field definitions and no data records. From this point on, the other NDB command options and the &NDBxxx verbs can refer to it.

Start the Database

To allow NCL access to the database, the NDB START command must be used to keep the database active:

```
NDB START dbname [DEFER | NODEFER] [LOADMODE]
```

If the database is to be bulk-loaded (described in Task 9), consider using the DEFER option of the NDB START command. This option tells the database manager not to flush buffers after each update command (including &NDBADD, &NDBDEL, &NDBUPD), which improves performance at the expense of integrity. The DEFER option of the NDB START command is effective only if the database was opened with the UDBCTL command using the LSR and DEFER options. The LOADMODE option can be used to indicate that bulk-loading is to occur.

Insert Field Definitions

An NDB cannot be used without field definitions. Field definitions can be added to or deleted from an NDB, using the NDB FIELD command at any time. Just after the successful completion of an NDB CREATE followed by an NDB START is a good time to add field definitions.

Example: Insert Field Definitions

The following example shows how to add field definitions:

```
NDB FIELDdbname ADD=SURNAMEFMT=C KEY=Y +
NULLFIELD=N NULLVALUE=N
NDB FIELDdbname ADD=FIRSTNAMEFMT=C KEY=N
NDB FIELDdbname ADD=DOBFMT=D KEY=N
NDB FIELDdbname ADD=ADDR1FMT=C KEY=N
NDB FIELDdbname ADD=ADDR2FMT=C KEY=N
NDB FIELDdbname ADD=ADDR3FMT=C KEY=N
NDB FIELDdbname ADD=ADDR4FMT=C KEY=N
NDB FIELDdbname ADD=SEXFMT=C KEY=N
NDB FIELDdbname ADD=NAME
```

Note: The definitions could also have been added by using the &NDBDEF verb.

If the database is to have a sequence key, then the definition for it must be added first. A sequence key is defined by specifying KEY=SEQUENCE (can be abbreviated to KEY=S) on the field definition. A sequence key is forced to have the attributes UPDATE=NO, and NULLFIELD=NO. Records must always have a unique value for the sequence key field (like KEY=UNIQUE). A sequence key field definition cannot be deleted.

Load Initial Data

If the database needs to have data loaded into it (for example, a table or reference database), then the data can be loaded using an NCL procedure to read the input data (for example, from a UDB, either a KSDS or an ESDS), and use &NDBADD to add it to the NDB.

If a large amount of data is to be loaded this way, then use the DEFER option of the NDB START command to prevent buffer flushing during the load.

When the load completes successfully, use an NDB START NODEFER command to remove the defer status. There is no need to stop and restart the database.

Another way to speed up loading of large amounts of data is to create or start the database in LOAD MODE. In this case, no keys are manipulated while loading, making the load run much faster.

The NDB ALTER command must then be used to build all keys in a single pass.

Note: If a region failure occurs while an NDB is open in DEFER mode, then it is flagged as unusable, and must be restored or recreated.

If the database is in LOAD MODE, it must have an NDB ALTER command run against it to build keys.

The load program should use the EXCLUSIVE option of the &NDBOPEN statement to prevent other users accessing the database while it is running.

The progress of the load program can be monitored by periodic use of the SHOW NDB=dbname command. This command displays information about the database, including the number of database requests executed (this is the sum of NDB commands and &NDB NCL statements) since it last started.

This completes the process of initialization (and initial loading) of an NDB. The tasks are the same as required for any other UDB, except for the NDB CREATE and field definition tasks.

Most of the initialization tasks can be combined into one NCL procedure, using &INTCMD/&INTREAD to issue and check the commands, and &NDBxxx statements to perform the definition and load. In a z/OS or MSP environment, UTIL0007 can be used to perform the VSAM DELETE/DEFINE from NCL.

Delete an NDB

An NDB can be deleted by using the standard VSAM (IDCAMS) DELETE command:

```
DELETE clustername CL
```

If the NDB is in use, then it must first be closed. To accomplish this, issue the following commands:

```
NDB STOP dbname IMM LOCK
UDBCTL CLOSE=dbname
UNALLOC DD=dbname
```

The NDB STOP command, with the IMM and LOCK operands, immediately stops the database, if it is active, and locks it from further access by any NDB commands or &NDBxxx statements. Any currently signed on users are given response 250 on their next request.

The UDBCTL CLOSE command physically closes the data set.

The UNALLOC command frees the data set so that it can be deleted.

Following the successful completion of the commands, the IDCAMS DELETE can then be issued.

If the NDB is being deleted in preparation for reuse as an empty database, then the NDB RESET command provides a more convenient way to do this. Alternatively, if the data set was defined to VSAM with the REUSE option, then a UDBCTL OPEN RESET command causes VSAM to clear the data set back to empty.

Delete All Data in an NDB

If an NDB is being used as a journal file (that is, it is cleared regularly after a specified time or number of records, for example), it can be cleared in three ways:

- Write an NCL procedure that reads the entire database sequentially (for example, by RID) and deletes all records-this method is slow and tedious.
- Physically delete the data set (or issue a UDBCTL OPEN RESET command if the data set is defined with REUSE), and re-create it, as described previously-this method has practical uses, such as when the data set must be relocated on DASD or needs more space.
- Use the NDB RESET command-this method is normally the best approach.

  The NDB RESET command deletes all data records (and their keys) from an NDB, but preserves the field definitions. Thus, it is equivalent to deleting, defining, re-creating, and reissuing all &NDBDEF ADD statements or NDB FIELD commands required to build the field definitions.

  An NDB must not be active when the NDB RESET command is issued. This prevents active users from having the database cleared underneath them. A LOCKED database cannot be reset.

  The following sequence of commands resets an NDB:

  ```
  NDB STOP dbname IMM
  ```

  ```
  NDB RESET dbname
  ```

  If the database is not active, then the NDB STOP command effectively does nothing. The RESET command gives you the option of placing the database into LOAD MODE.

Alter Field Definitions in an NDB

Altering field definitions can be broken into three activities:

- Adding new field definitions to an NDB
- Deleting field definitions from an NDB
- Updating field definitions in an NDB

The first two can be done directly, at any time, even while users are accessing the database. Use the NDB FIELD ADD= fieldname and NDB FIELD DELETE= fieldname commands.

Add Field Definitions

Adding a field definition makes the field immediately available to all users.

When adding field definitions after data is already in an NDB, defining a field with NULLFIELD=NO causes errors on the update of any record that existed prior to the definition, if the newly defined field is not included in the update list. This is because it is a required field, and is not in the record.

Delete Field Definitions

Deleting a field definition logically removes the field from the database. Following the delete, all data values for that field become inaccessible.

Deleting a field immediately makes the field inaccessible to NCL procedures. Any currently active sequences (&NDBSEQ) defined on that field, if it is keyed, are given a response code on the next &NDBGET referring to that sequence. Predefined formats (&NDBFMT) referring to the field still valid until they are redefined. An in-progress &NDBSCAN could get undefined results.

Deleting a field causes a physical VSAM delete to be performed for all the associated key records. The data records, however, are not updated. To do so would create an unacceptable overhead. Instead, the field value in each data record is regarded as logically deleted. It is inaccessible. Whenever a data record is updated, all logically deleted fields are removed.

Update a Field Definition

A field definition can be updated at any time. However, not all field attributes can be changed at any time. Some changes require the database to be empty (that is, just created or reset), or in LOAD MODE. Some field attribute alterations are prohibited.

The following field attributes can be changed at any time:

- DESC = description
- USER1, USER2, USER3, USER4
- NULLVALUE
- NULLFIELD (except for KEY = SEQ field)
- UPDATE (except for KEY = SEQ field)
- KEY = Y to KEY = N
- KEY = U to KEY = Y or N
- NEWNAME (to rename the field)

The following attribute can be changed if the field is not keyed (that is, KEY=NO):

- CAPS = YES to CAPS = SEARCH

The following attributes can be changed if the database is in the LOAD MODE or empty:

- KEY (except to/from SEQ)
- CAPS = SEARCH/NO to CAPS = NO/SEARCH

The following attributes can be changed if the database is empty:

- FMT
- CAPS
- KEY (except to/from SEQ)

The following attribute cannot be changed:

- KEY = SEQ (to or from)

Note: A field can also be changed from KEY = N to KEY = Y by using the NDB ALTER command.

Back Up an NDB

Back up an NDB with the standard IDCAMS utility functions (that is, REPRO or EXPORT). For integrity, the data set should not be open. Issue the following commands to ensure this:

```
NDB STOP dbname IMM LOCK
UDBCTL CLOSE=dbname
```

Following completion of the backup, issue the following command to make the NDB available again:

```
UDBCTL OPEN dbname options
NDB START dbname UNLOCK options
```

The options on these commands should agree with any standard options specified (for example, LSR/DEFER on the UDBCTL statement).

Restore an NDB

If an NDB must be restored from a backup copy, then use the standard IDCAMS restore facility associated with the backup copy. For example, if REPRO was used to backup the database, then use REPRO to restore it; if EXPORT was used to backup the database, then use IMPORT to restore it.

When using REPRO, define the database with REUSE to allow the REUSE option of REPRO to be used. This makes a DELETE/DEFINE prior to the restore unnecessary.

The database must not be open while a restore is in progress. To create a consistent environment for NCL procedures using the NDB, issue an NDB STOP LOCK command prior to the restore. This causes all &NDBOPEN statements executed to get a database locked response.

Upon completion of the restore, issue the UDBCTL OPEN, as required, and an NDB START UNLOCK command to restart the database, and release the lock.

Monitor NDB Activity

Use the SHOW NDB, SHOW NDBUSER and the TRACE option of the NDB START command to monitor NDB activity.

To determine the number of NDBs currently active, issue the command:

```
SHOW NDB
```

The output from this command shows the number of NDBs active, locked, and stopping. To obtain detailed statistics about all active or locked NDBs, or a specific NDB, use the command(s):

```
SHOWNDB=ALL-* for all, or
SHOWNDB=dbname-* for a specific NDB.
```

The output from these commands indicates the status of the NDB(s), the number of signed on users, the number of commands processed, and whether the database was started in DEFER status.

This display is particularly useful for monitoring the progress of long-running procedures that are loading or reading large numbers of records.

To determine the user ID(s) of users signed on to all active NDBs, issue the command:

```
SHOW NDBUSER
```

The output from this command, which is similar to the SHOW UDBUSER command, indicates details such as the database, user ID, and LU name (terminal) for all users signed on to any NDB.

Note: A SHOW UDBUSER shows *NDB as the only user of a UDB that is an NDB. This serves as an indication to the issuer of the SHOW UDBUSER command that a SHOW NDBUSER command is required for this UDB/NDB to obtain a list of signed on users.

Using the TRACE option of the NDB START command causes a message to be written to the activity log every time an &NDB verb or an NDB command is issued involving that particular NDB. Tracing can be stopped at any time by issuing an NDB START command specifying NOTRACE.

Monitor NDB Performance

Monitoring is used, for example, for physical tuning and buffer tuning. Use a periodic IDCAMS LISTCAT command to monitor the physical attributes of the NDB, including DASD space, number of extents, and CI splits.

An NDB itself does not need reorganization internally. VSAM, however, might need to reorganize to remove excessive CI and/or CA splits if a large amount of key addition, deletion, or value changing takes place. This can be accomplished by using standard IDCAMS services to backup, delete, redefine, and restore the data set.

Important! You must not change the VSAM key length or maximum record length during this reorganization. If you do, the NDB will not be usable.

If you need to increase the key length, the NDB must be logically unloaded, deleted, recreated, and logically reloaded.

The SHOW VSAM command allows you to determine whether the NDB performance would benefit from increased buffering, if not running from the LSR pool, or from increases or changes in the LSR pool definition. Standard VSAM tuning techniques should be followed.

Improve Performance by Using LOAD MODE

As NDBs use inverted-list indexes, bulk record addition, update, or deletion can be slow. This is because of the large number of physical file updates required to manage all the keyed field indexes.

To allow fast loading (in particular), an NDB can be placed in LOAD MODE. In LOAD MODE, no keys (other than the optional sequence key) are maintained. This greatly improves record add speed.

When in LOAD MODE, records can be added, updated, deleted, retrieved, or scanned. However, none of these operations use keys. This means that GET or SEQ by a key is not allowed. Only RID access is permitted.

The only way to take an NDB out of LOAD MODE is to use the NDB ALTER command. This command, using the BLDX DB option, reads all the NDB data records, extracts all keys, sorts them, and writes them in a single pass. It checks for errors (for example, unique key violation) and, when finished, resets LOAD MODE.

Check an NDB for Consistency

If you want to validate the relationship between keys and data in an NDB, you can use the NDB ALTER command CHKX option to do this. It extracts and sorts all keys from the data and then compares them with the actual key records in the NDB. All errors are reported.

Multiple System Access to an NDB

An NDB may be corrupted by:

- The database being opening by two regions (same machine or shared DASD)
- The database being opened twice (under different file IDs) on the same region

There are several internal protection mechanisms that attempt to prevent such corruption:

- The VSAM timestamp of the NDB is compared with all other open NDBs. A match prevents the NDB from opening. This blocks one NDB from being opened twice by the same region.

  It is not totally foolproof, as VSAM updates the timestamp each time the VSAM data set is physically closed (that is, UDBCTL CLOSE occurs).

  While an NDB is open, the control record contains the domain ID of the accessing product region. This field is closed when the NDB is closed.

  If, when opening an NDB, a non-blank domain ID is found, and it is not the same as the current domain ID, then the NDB does not open unless the FORCE operand of the NDB START command is used.

  This prevents a database being opened by two different regions. Use the FORCE operand if there is a system failure and the NDB must be opened on a different region.

- The name used to open the NDB (that is, UDBCTL ID = operand value) is stored in the NDB control record. If a mismatch occurs on open, a Important! message is written. The open is allowed to continue unless other problems occur.
- Despite the previous precautions, an NDB can always be opened in INPUT MODE. This bypasses all the checks but no updates can be performed.
- NDB security-NDBs can be protected by the NCL security exit, NCLEX01 (see page 517). This protection can be used to restrict users' NCL procedures to specific NCL statements for functions such as updating and adding.

How to Implement NDB Journaling

To implement NDB journaling, perform the following steps on your region:

1. Define two journal data sets (see page 312)
2. Allocate the data sets to the product region (see page 312)
3. Make duplicates of the NDBs (see page 313)
4. Add the duplicates to the batch forward recovery JCL (see page 313)
5. Start the NDBs (see page 314)

Define Two Journal Data Sets

Use IDCAMS to define two VSAM entry-sequenced data sets (ESDS). The following parameters are required (use sample $NDIDCDJ):

NONINDEXED

Indicates an ESDS.

REUSE

Journal is cleared when opened for update.

RECORDSIZE

Journal record size should be greater than the record size minus key length + 60 of the largest NDB using the journal.

The space allocated to the journal depends on the number of NDB updates occurring on the system, and the frequency with which you swap journals for batch processing. The volume should ideally be on a separate unit to the NDBs being journaled.

Allocate Data Sets to the Product Region

Allocate the two journal data sets, using DISP=SHR, to the product region by either including them in the JCL, declaring them in the NETMASTR SYSIN deck, or allocating them in NMINIT or NMREADY NCL procedures.

Make Duplicates of the NDBs

Stop the NDBs you want to journal and create duplicates of them using IDCAMS REPRO.

The duplicate must be created after the NDB CREATE has been performed to initialize the NDB.

Once the duplicate has been created, it is important to keep it synchronized with the primary copy by ensuring that all NDB journal records written after the duplicate was made are applied to it.

For an existing NDB which has already been loaded, the decision to start journaling updates can be made at any time.

If you regularly shut down your product region, CA recommends that you resynchronize your NDB backups with the primary copy. Do not apply journals created before the new backup copy.

Add the Duplicates to the Batch Forward Recovery JCL

The batch forward recovery utility (UTIL0010) applies the NDB journal to a duplicate copy of the NDBs.

Add the data set name of the NDB duplicate to your batch forward recovery JCL (use sample $NDUT010).

Make sure you have updated the JOURNL1 and JOURNL2 DD statements in the sample with the two journal data set names you defined in Task 1.

Include in the utility JCL all NDBs which require forward recovery. By default, the utility attempts to forward recover all NDBs which have journal records present in the journal data set. The DD name for the NDB should be the same as the NDB name used to identify it to the product region.

Use the control cards on the U10IN data set to specify a subset of NDBs.

The format of a U10IN control card is:

```
RECOVER NDB=ndbid [,DD=name]
```

where the DD operand can be used to specify an override DD name for the NDB.

RECOVER ALL is used to indicate that all NDBs are to be forward recovered. This is the default.

The following is a sample JCL specification for the forward recovery utility:

```
//FWDRECVR EXEC PGM=UTIL0010,PARM='JOURNAL=JOURNL1'
//STEPLIB DD DSN=steplib,DISP=SHR
//U10PRINT DD SYSOUT=*
//U10IN DD*
  RECOVER ALL
//JOURNL1 DD DSN=USER.NDB.JOURNAL1,DISP=SHR
//JOURNL2 DD DSN=USER.NDB.JOURNAL2,DISP=SHR
//*
//NDB1 DD DSN=USER.NDB.BACKUP.NDB1,DISP=SHR
//NDB2 DD DSN=USER.NDB.BACKUP.NDB2,DISP=SHR
//NDB3 DD DSN=USER.NDB.BACKUP.NDB3,DISP=SHR
//*
```

Note: The JOURNAL=JOURNL1 parameter specified on the EXEC UTIL0010 statement is used to override the journal DD name. If this parameter is not specified, then the DD name used is JOURNAL.

Start the NDBs

Issue an NDB START command with the JOURNAL operand specified.

The first time an NDB START command (with the JOURNAL operand) is issued, it causes the journal data sets to be opened. The system normally checks both journal data sets and begins processing with the oldest one. This gives you time to run the forward recovery utility if the system was restarted after failure.

If JOURNL1 is empty, then no attempt is made to check the second data set, and journaling begins immediately on JOURNL1.

NDB Journal Swapping

Journals swap if a JOURNAL SWAP command is issued or if the journal in use runs out of space.

Each time a journal is swapped, the journal control NCL procedure is started to assist the automation of forward recovery. The SYSPARMS JRNLPROC command can be used to specify the procedure name (the default is $NDJPROC).

If the system terminates abnormally, submit the NDB forward recovery job manually (or from an alternate automation process), as the other journal is used after the restart, and a swap effectively occurs.

Forward recovery must be performed before a journal swap occurs after the system restart.

We recommend that you apply journals to the NDB backups as soon as they are swapped.

Chapter 17: Using &NDB Verbs

This section provides examples of the use of the NCL verbs related to NetMaster Databases (NDBs). There are examples that include adding, deleting, updating, and retrieving records, as well as examples that use information about the database.

The success or failure of many &NDB verbs is indicated as a completion or error code returned in the system variables &NDBRC and &NDBERRI on completion of the function. Throughout this section, reference is made to specific return codes that can occur on particular conditions.

This section contains the following topics:

Relationship Between &FILE and &NDBxxx Verbs

Although the &NDB verbs are functionally similar to the &FILE verbs, there are some significant differences.

More information:

Differences Between NDBs and UDBs (see page 285)

Protect Your Data Values with &NDBQUOTE

Several &NDB verbs use a free-form text syntax.

For more information about the &NDBSCAN verb syntax, see the *Network Control Language Reference Guide*.

When processing the free-form text, certain characters, for example, '(', ',' and ')', act as delimiters. If the data value contains one of these characters, it could cause a syntax error, and will not be preserved.

To prevent this, data values can be quoted, using either single (') or double (”) quotes. If the data contains the selected quote character, two of that character must be used.

The &NDBQUOTE built-in function automatically quotes the data when required. It knows the characters that require quoting, and picks single or double quotes as required. It also handles doubling of embedded quotes.

Preserve Lowercase Data

If the data you store in an NDB contains lowercase characters, ensure that &CONTROL NOUCASE is in effect in all procedures that use the &NDBADD or &NDBUPD verbs. Failure to do so could mean the accidental folding of data to uppercase. This requirement also applies to &NDBDEF and the NDB FIELD command when adding field descriptions or USER1–4.

Define and Delete Fields in an NDB

An NDB has no predefined fields. All fields must be defined once before they can be used. This is normally done by the Database Administrator when the NDB is defined.

Field definitions can be added or deleted at any time, even while other users are accessing the NDB, although you might not want to do this.

The names and attributes of defined fields can be obtained by using the &NDBINFO verb.

More information:

Alter Field Definitions in an NDB (see page 305)
Obtain Information About an NDB (see page 331)

Access an NDB

Before any records can be added to, deleted from, or retrieved from an NDB, the NCL process must sign on, or connect itself to the NDB. This is accomplished using the &NDBOPEN verb:

```
&NDBOPEN dbname
```

This statement registers the NCL process as a signed-on user of the nominated NDB.

If you want the NCL process to have exclusive control over the NDB, then specify the keyword, EXCLUSIVE, after the database name.

If you want to perform input only operations, then use the INPUT keyword.

If a security exit is being used, the DATA keyword can be used to allow up to 50 characters of data to be passed to the exit.

The &NDBRC system variable should always be checked after &NDBOPEN, to ensure it was successful. If the value of &NDBRC is not 0, then the open has failed, except for response 34, which indicates that the NCL process is already signed on.

When an &NDBOPEN is executed, &NDBCTL ERROR=CONTINUE is always assumed, to allow the NCL process to handle errors at this point. If an open failure occurs, and no action is taken, then the next &NDBxxx statement for that database will get a not open response. Response 34 (already open) will terminate the procedure unless an explicit &NDBCTL ERROR=CONTINUE is in effect.

The &NDBOPEN verb is analogous to the &FILE OPEN verb. However, there is no need to reissue &NDBOPEN after referencing an NDB of another name. This is because &NDB statements that refer to an NDB require the name of the NDB that is to be accessed, whereas the &FILE verbs assume the UDB named in the last executed &FILE OPEN statement.

EASINET Considerations

If NDBs are being accessed by EASINET procedures, they should not be open across the &PANEL statement that displays the network logo. To do so would increase the amount of storage each network terminal would need.

Close an NDB

When an NCL process has completed use of an NDB, use the &NDBCLOSE verb to sign off (disconnect) the NCL process from the NDB:

```
&NDBCLOSE dbname
```

The &NDBCLOSE verb disconnects the NCL process from the nominated NDB, if it is connected (signed on).

All storage associated with the process is freed. This includes storage for defined formats (&NDBFMT), sequences (&NDBSEQ), and scan result lists (&NDBSCAN).

&NDBCLOSE is analogous to the &FILE CLOSE verb.

Work with NDBs

You can use NCL verbs to add records to and update, delete, and retrieve them from an NDB.

Add Records to an NDB

The &NDBADD verb is used to add records to an NDB. The fields for the new records must be named on the &NDBADD statement, along with their values. Not all fields need to be supplied. For the database manager, only those fields with the attribute NULLFIELD = NO need be supplied-this includes the sequence key, if one is defined.

If you only need to supply a few fields on the &NDBADD statement, code it as follows:

```
&NDBADD dbname DATA name1 = value1 name2 = value2 ...
```

If you need to supply a large number of fields, or you are not sure of the exact content of the new record (for example, table driven systems), the second format of the &NDBADD verb can be used:

```
&NDBADD dbname START
&NDBADD dbname DATA name1 = value1
&NDBADD dbname DATA name2 = value2
&NDBADD dbname DATA name3 = value3
&NDBADD dbname DATA name4 = value4
&NDBADD dbname END
```

This syntax allows a record of any length to be created.

The fields need not be in any order, and the collection of &NDBADD statements need not be adjacent. For example, a loop could be used, with complex substitution, to build the list of field names and values:

```
&NDBADD MYNDB START
&I = 1
&DOWHILE &I LE &NFLDS
   &VALUE = &NDBQUOTE &FV&I
   &NDBADD MYNDB DATA &FN&I = &VALUE
   &I = &I + 1
&DOEND
&NDBADD MYNDB END -* this statement calls the DBMS
```

In this example, you loop through a table of field names and values, adding each field.

Following the &NDBADD, or &NDBADD END, if you are using START/DATA/END, the &NDBRC system variable contains 0 if no errors were encountered. If not 0, an error response indicates the problem, and &NDBERRI might have additional information. An error message is also displayed unless &NDBCTL MSG=NO is in effect.

If the response is 0, &NDBRID contains the RID assigned to the new record. This RID can be used in other &NDB statements to refer to the new record.

Update Records in an NDB

A record in an NDB can be updated using the &NDBUPD verb. Unlike the &FILE PUT verb, an &NDBUPD verb only updates the nominated fields. All other fields retain their existing value. The &FILE PUT verb replaces the entire record.

The RID(s) of the record(s) to be updated must be known. Normally, the RID is determined by a preceding &NDBGET or &NDBSCAN.

To update just a few (fixed amount) fields, code:

```
&NDBUPD dbname RID=rid DATA name1 = value1 name2 = value2
```

where name1, name2, ... are the names of the fields to be updated, and value1, value2, ... are the values.

If you need to update a large number of fields, or you are not sure of the exact content of the update, an alternative format of the &NDBUPD verb can be used:

```
&NDBUPD dbname RID=n START
&NDBUPD dbname DATA name1 = value1
&NDBUPD dbname DATA name2 = value2
&NDBUPD dbname DATA name3 = value3
&NDBUPD dbname DATA name4 = value4
&NDBUPD dbname END
```

This syntax allows any number of fields to be updated.

The fields need not be in any order, and the collection of &NDBUPD statements need not be adjacent. For example, a loop could be used, with complex substitution, to build the list of field names and values:

```
&NDBUPD MYNDB RID=&UPDRID START
&I = 1
&DOWHILE &I LE &NFLDS
   &NDBUPD MYNDB DATA &FN&I = &VALUE
   &I = &I + 1
&DOEND
&NDBUPD MYNDB END -* this statement calls the DBMS
```

In this example, we loop through a table of field names and values, updating each nominated field.

Note: Fields defined with UPDATE=NO can be specified in the update list, as long as the same value as is currently in the record is supplied.

A field that is to be set to null (not present) can be represented by coding:

```
fieldname = &NULL
```

where &NULL is an undefined variable. If the field definition has NULLFIELD=NO, an error response will be given.

Following the &NDBUPD or &NDBUPD END, if you are using START/DATA/END, the &NDBRC system variable contains 0 if no errors were encountered. If not 0, an error response indicates the problem, and &NDBERRI might have additional information. An error message is also displayed unless &NDBCTL MSG=NO is in effect.

If the response is 0, &NDBRID contains the RID that was supplied on the &NDBUPD verb.

Delete Records from an NDB

To delete records from an NDB, use the &NDBDEL verb. The RID(s) of the record(s) to be deleted must be known. The RID is normally obtained from a preceding &NDBGET or &NDBSCAN.

To delete a record, code:

```
&NDBDEL dbname RID=&rid
```

where &rid is an NCL variable that contains the RID.

There is no direct equivalent to the &FILE DEL KEQALL/KGEALL generic delete. However, embedding an &NDBDEL in a loop controlled by an &NDBGET GENERIC or &NDBGET SEQUENCE=name allows easy deletion of any group of records (particularly powerful after an &NDBSCAN).

Example: Delete Records from an NDB

This example shows how to delete all records that have SURNAME = SMITH and STATUS = DEPARTED.

```
&NDBSCAN MYNDB SEQ=S1 DATA SURNAME = SMITH AND STATUS = +
                     DEPARTED
&IF   &NDBRC = 0 &DO
   &NDBGET MYNDB SEQ=S1 FORMAT NO-FIELDS
   &DOWHILE &NDBRC = 0
      &NDBDEL dbname RID=&NDBRID
      &NDBGET dbname MYNDB SEQ=S1 FORMAT NO-FIELDS
      &DOEND
&DOEND
```

Following the &NDBDEL, the &NDBRC system variable is set to 0 if the delete was successful, or 1 if no record with the supplied RID was found.

Retrieve Records from an NDB

To retrieve records from an NDB, use the &NDBGET verb. There are two related verbs:

- &NDBFMT, used to predefine a list of the fields to be returned by &NDBGET
- &NDBSEQ, used to define a sequential read path for an NDB

You can retrieve records from an NDB in several ways:

- Direct by RID, allowing EQ, GE, GT, LE, LT relationships with your RID value. This is useful when you know the RID of the record you want.
- Direct by any keyed field. The key field value of the returned record can be EQ, LE, LT, GE, GT, or generically equal to the supplied key. If the key field is not unique, only the record with the lowest RID for any set of records with the same key field value can be accessed this way.
- Sequentially by RID. Records are returned in ascending or descending RID sequence. For databases without a sequence key, this is the fastest way to read the entire database. Optionally, a starting and/or ending RID can be nominated. The last record read can also be re-read without knowing its RID.
- Sequentially by any key field (including a sequence key, if defined). Records are returned in ascending or descending key field sequence. If the key is not unique, RID is used to sequence records with an equal key field value. For databases defined with a sequence key, retrieval by that field is the fastest way to read the entire database. Optionally, a starting and/or ending key value can be nominated. The last record read can also be re-read without knowing its RID.
- Sequentially, from a list of records that passes an &NDBSCAN.
- Indirectly by a keyed field where its value is stored in another record in the NDB.
- You can also retrieve statistical information about any key field (also known as a histogram).

More information:

Scan Processing (see page 345)

Define Fields to Return (&NDBFMT)

When retrieving records from an NDB, it is likely that not all the fields in a record are needed. Also, it might be desirable to return the fields in NCL variables with different names. The &NDBFMT verb lets you define any number of named formats, that can be nominated on an &NDBGET statement, referring to the same NDB. The nominated format determines which fields will be returned, and the names of the NCL variables that will receive the values.

The defined format can request that all fields defined in the database be returned (ALL-FIELDS), or that all fields defined as keyed be returned (KEY-FIELDS), or that no fields be returned (NO-FIELDS) (useful for just checking for existence of a record), or that a list of nominated fields be returned, optionally in NCL variables with a different name.

Note: A format can also be specified directly on the &NDBGET statement, but this is not recommended, particularly if the &NDBGET statement is in a loop. The format definition must be parsed and encoded every time the &NDBGET statement is executed.

A format stays defined until explicitly deleted (by &NDBFMT ... DELETE), or until an &NDBCLOSE is executed for that NDB.

The following are examples of predefined formats:

```
&NDBFMT MYNDB DEFINE FORMAT=FMT1 DATA ALL-FIELDS
&NDBFMT MYNDB DEFINE FORMAT=FMT2 DATA NO-FIELDS
&NDBFMT MYNDB DEFINE FORMAT=FMT3 START
&NDBFMT MYNDB DATA FIELDS
&NDBFMT MYNDB DATA SURNAME FIRSTNAME
&NDBFMT MYNDB DATA ( LNAME = LASTNAME)
&NDBFMT MYNDB END
```

It is evident that the START/DATA/END syntax, as described under &NDBADD and &NDBUPD, is also available when using &NDBFMT. Thus, complex variables and open-ended formats can be defined. The START/DATA/END syntax is not available on the &NDBGET verb. Any format defined in an &NDBGET statement must fit on the &NDBGET statement itself.

The examples all assume predefined formats.

Determine Which Fields Are Present

Since NDBs support null fields, it might be necessary to determine, when a particular record is read, which fields are present and which are not. The MODFLD option on the &NDBGET verb allows the use of &ZVARCNT and &ZMODFLD to access the modified fields:

```
&NDBGET dbname retrieval-method format-method MODFLD=YES
&field1 = &ZMODFLD
&field2 = &ZMODFLD
...
```

Retrieve Records Directly by RID

When you know the RID of the record that you want, perhaps from an earlier &NDBADD, you can retrieve it by coding:

```
&NDBGET dbname RID=ridvalue FORMAT=fmtname
```

ridvalue is a variable containing an RID obtained elsewhere.

You can also code OPT=KGE, and so on, to retrieve a record with an RID satisfying the coded option. For example, OPT=KGT returns the record with the next-highest RID than the value passed. It is possible to simulate sequential reading by RID using this technique.

Retrieve Records by Key Field

When you know the value for a key field, retrieve a record matching that key field by coding:

```
&NDBGET dbname FIELD=fieldname VALUE=value FORMAT=fmtname
```

Code OPT=KGE to return a record where the key field satisfies the coded option. For character or HEX fields, OPT=GENERIC is also supported. Use OPT=KGT for forward, or OPT=KLT for backward, and supply the last key field value as the search argument to simulate sequential reading.

For key fields that are not unique, this technique always returns the record with the lowest RID, from the set of records having the same key field value. The other records cannot be accessed this way.

Read Sequentially by RID

When the entire database, or a large portion of it, must be read sequentially, and the order of returned records is unimportant, the following technique should be used.

```
&NDBSEQ dbname DEFINE SEQ=SEQ1 RID
&NDBGET dbname SEQ=SEQ1 FORMAT=fmtname
&DOWHILE &NDBRC = 0
   ...  process record
   &NDBGET dbname SEQ=SEQ1 FORMAT=fmtname
&DOEND
```

The &NDBSEQ statement sets up a sequential retrieval definition, that can be used on an &NDBGET statement to retrieve records sequentially by RID. By default, the options SKIP=+1 and DIR=FWD are assumed on the &NDBGET, so the database is read from lowest RID to highest RID.

Options on the &NDBSEQ statement allow specification of a low and high RID.

If the database has a sequence key, the next method will perform better.

Retrieve Records Sequentially by Key Field

When you want to retrieve records by any key sequence, the following technique can be used:

```
&NDBSEQ dbname DEFINE SEQ=seqname FIELD=keyfieldname +
FROM=fromvalue TO=tovalue
&NDBGET loop as coded in previous example.....
```

This use of &NDBSEQ and &NDBGET allows sequential retrieval by any key field, and, for non-unique key fields, records with the same key field value are returned in RID sequence within the value.

The FROM and TO operands on &NDBSEQ are optional. If omitted, the lowest (FROM) or highest (TO) key values are used. For non-unique key fields, VALUE=value can be coded to indicate the from and to values are the same. This is useful when you want to read all records with the same non-unique value. For character and HEX fields, GENERIC=value can be coded to indicate the range is to cover all records with that generic key value.

Retrieve Records Indirectly by Key Field Stored in Another Record

If the NDB has multiple record types (see page 291), the key identifying the desired record is not always known immediately. The identifying key might be stored in the field of another record.

For example, if the NDB contains an ordering system, it might be necessary to access a customer record, given only an order number, the order record of which contains a key identifying a separate customer record.

NDBs allow this type of access through the .LINK option of the &NDBGET/&NDBFMT verbs. The technique is:

```
&NDBGET dbname RID=record ID FORMAT FIELDS .LINK +
(FROM=fldname TO=keyed-fieldname ) FIELDS formatlist
```

Up to 16 different records can be accessed in the one &NDBGET statement using linking; that is, one primary and up to 15 secondary records.

Any access method can be used for retrieving the first record. The retrieval method for subsequent records is equivalent to an OPT=KEQ get.

Retrieve Keyed Field Statistics (Histogram)

All keyed fields in an NDB (except the sequence key) contain statistical information on the number of records that have a particular unique key value.

The KEY option of &NDBGET and &NDBSEQ allow retrieval, not of a record, but of a key value and record count. Thus, statistical information can be obtained. This is very fast, as no data is accessed. When reading the NDB in this way, the format specified on &NDBGET is completely ignored. (You can, for example, code FORMAT = NO.) Instead, the data is returned in two NCL variables:

```
&NDBKEYVALUE keyed field value
```

```
&NDBKEYCOUNT keyed field record count
```

For example, to find out how many records have field surname with a value of 'Smith':

```
&NDBGET dbname KEY=SURNAME VALUE='SMITH' FORMAT=NO
```

To get a sequential breakdown of a key field's contents and counts:

```
&NDBSEQ dbname DEFINE SEQ=S1 KEY=SURNAME
&NDBGET dbname SEQ=S1 FORMAT=NO
&DOWHILE &NDBRC=0
     &WRITE &NDBKEYVALUE &NDBKEYCOUNT
     &NDBGET dbname SEQ=S1 FORMAT=NO
&DOEND
```

Notes on Sequential Retrieval

The &NDBSEQ verb and the &NDBSCAN verb let you specify a name for a sequence. Unlike the &FILE verbs, an NCL process can have any number of concurrent sequences defined on any number of databases. Positioning is handled internally, with no need to use accompanying number of VSAM strings.

The only overhead is an internal NDB control block that maintains positioning.

Performing a direct get does not destroy any positioning maintained by active sequences.

KEEP=YES on &NDBSEQ

By default, a sequence is automatically deleted when an end-of-file response is returned (&NDBRC = 2). This is normally what is wanted, and saves you coding an &NDBSEQ DELETE. If, however, you want the sequence to stay defined, you can code KEEP=YES on the &NDBSEQ DEFINE, and the sequence stays defined until explicitly deleted, or the NDB is closed.

&NDBSEQ RESET

A defined sequence can also be reset, allowing you to restart the sequence at any point. For example, after reading forward, and getting an EOF response, you are positioned after the last record in the sequence. Forward gets will continue to return the EOF response. To re-read from the beginning, issue an &NDBSEQ RESET as shown:

```
&NDBSEQ dbname RESET SEQ=seqname
```

Optionally, you can nominate a key value to reposition to. The next get will return the first record with the matching, or higher, or lower, if no match, and depending on the direction:

```
&NDBSEQ dbname RESET SEQ=seqname REPOS=value
```

&NDBGET DIR= and SKIP=

When reading sequentially, the &NDBGET verb lets you specify the direction of retrieval, FWD or BWD, and a skip amount.

The default is DIR=FWD and SKIP=+1, which causes the next-higher key record to be read.

SKIP=0 causes a re-read of the last record read in that sequence. This is useful where you are retrieving only a subset of fields, and sometimes need to obtain extra fields. You need not specify a direct get by RID.

A skip amount (n) greater than 1 will allow you to skip n-1 records on each &NDBGET. This is extremely useful when extracting samples from a database, and especially useful when scrolling in selection lists (see the following examples).

A negative skip amount causes the specified or defaulted direction to be inverted. For example, SKIP=-5 is equivalent to SKIP=5 with DIR=BWD.

Specifying DIR=BWD allows a backward read to be performed. Thus, when you need to retrieve records in descending key sequence, DIR=BWD can be used.

Read by Sparse Keys

When defining a sequence on a key field that is defined (or defaulted) with NULLFIELD=YES, it is possible that some records do not contain the field. These records will not be returned when reading by that sequence. This is because no index record is built for fields not present in a data record. This can be especially useful when the database contains multiple record types, as illustrated in one of the following examples.

Example 1: Read by Sparse Keys

This example shows how two sequences defined on a single database, where the logical record types are disjoint, using different key fields, can perform a master/transaction update.

```
&NDBSEQ MYNDB DEFINE SEQ=MAST FIELD=MASTKEY
&NDBSEQ MYNDB DEFINE SEQ=TRAN FIELD=TRANKEY
&GOSUB .READMAST  -* read mast, set &MASTKEY to 999999 if
                  -* eof
&GOSUB .READTRAN  -* read tran, set &TRANKEY to 999999 if
                  -* eof
&DOWHILE &MASTKEY.&TRANKEY NE 999999.999999
   &IF &MASTKEY = &TRANKEY &DO
            ...process match
      &GOSUB .READTRAN
   &DOEND
   &ELSE &IF &MASTKEY GT &TRANKEY &DO
            ...process unmatched transaction
      &GOSUB .READTRAN
   &DOEND
   &ELSE &DO
            ...process unmatched master
      &GOSUB .READMAST
   &DOEND
&DOEND
```

Example 2: Read by Sparse Keys

This example shows how to use SKIP= to extract a subset of a database, perhaps for statistical analysis.

```
&NDBSEQ  MYNDB DEFINE SEQ=S1 RID
&NDBINFO MYNDB DB    -* obtain # records in DB
&SKIP = &NDBDBNRECS / 1000 -* determine skip to get 1000
                     -* recs
&NDBGET  MYNDB SEQ=S1 SKIP=&SKIP FORMAT ALL-FIELDS
&DOWHILE &NDBRC = 0
               ...write sampled record.
   &NDBGET MYNDB SEQ=S1 SKIP=&SKIP FORMAT ALL-FIELDS
&DOEND
```

Example 3: Read by Sparse Keys

This example shows how to process a selection list using a sequence and SKIP/DIR.

```
&NDBSEQ MYNDB DEFINE FIELD=SURNAME KEEP=YES
&SKIPVAL = 1      -* initial skip
.LOOP &GOSUB .BUILD_PANEL  -* builds panel.
                  -* position now is last record on screen
&PANEL XYZPANEL
&IF &INKEY = PF08 &THEN &DO
   &SKIPVAL = +1  -* skip to next record after bottom
   &GOTO .LOOP
&DOEND
&ELSE       &IF &INKEY = PF07 &DO
&SKIPVAL = (0-(&LUROWS*2))
   -* skip top + back 1
-*screens length
   &GOTO .LOOP
&DOEND
```

Obtain Information About an NDB

The &NDBINFO verb lets you obtain information about the current state of an NDB, or any of the defined fields in an NDB.

To obtain information about the database, code:

```
&NDBINFO dbname DB
```

This statement sets several user NCL variables, prefixed by &NDBDB, with database information.

For more information about the fields, see the &NDBINFO verb description in the *Network Control Language Reference Guide*. Useful fields are:

&NDBDBNRECS

Contains the current number of records in the database.

&NDBDBNFLDS

Contains the current number of fields defined in the database.

&NDBDBVKL

Contains the VSAM key length of the database.

&NDBDBVRL

Contains the VSAM maximum record length of the database.

These fields can be used to determine if two NDBs are compatible (for example, for backup purposes).

The &NDBINFO verb also allows retrieval of information about fields defined in the database. Information can be retrieved about a field with a specific name, or by relative field number, where the number is from 1 to the value returned in &NDBDBNFLDS. This relative number is not fixed, but changes as field definitions are added and deleted. For this reason, inquiry by number should only be used in a loop (from 1 to &NDBDBNFLDS), with the database open EXCLUSIVE.

To retrieve information about a specific field, where the name is known, code:

```
&NDBINBFO dbname NAME=fieldname
```

To retrieve information about relative field number n, code:

```
&NDBINFO dbname NUMBER=n
```

In both cases, information is returned in several user NCL variables, prefixed by &NDBFLD. Following are several useful variables.

For more information about the returned variables, see the description of the &NDBINFO verb in the *Network Control Language Reference Guide*.

&NDBFLDNAME

Contains the name of the field (useful when retrieving by relative number)

&NDBFLDFMT

Contains the format of the field, for example, CHAR

&NDBFLDKEY

Contains the key option for the field, for example, UNIQUE

This information can be used in a table-driven database query and update program to edit or display data without needing to code specific details about each NDB. It is also useful for unload and reload programs.

Change NDB NCL Processing Options

The &NDBCTL verb is used to alter processing options associated with the executing NCL process. Options that can be changed are:

- Issuing messages on error conditions. By default (MSG=YES), a message is sent to the environment the NCL procedure is running under. For a normal NCL procedure, this is the OCS window. The messages are also logged. By specifying MSG=LOG, error messages are only sent to the activity log.

  By specifying MSG=NO, no error messages are issued. Only database integrity messages that are sent to monitor receivers are issued.

  Handling database error conditions. By default (ERROR=ABORT), a response code greater than 29 (in &NDBRC) causes the process to be terminated with an error message.

  By specifying ERROR=CONTINUE the process retains control, but must check &NDBRC for error responses.

  &NDBOPEN is always processed as if &NDBCTL ERROR=CONTINUE is in effect, except for response 34 (already open). The same applies to &NDBCLOSE, except for response 35 (not open).

- The format that dates can be entered (by &NDBADD/&NDBUPD, and &NDBSCAN), and returned (by &NDBGET). By default (DATEFMT=*), the user's UAMS language code, or, if no specific language code is in the user definition, the system language code, determines whether dates can be entered in DD/MM/YY (UK or DATE4), or MM/DD/YY (US or DATE5) format.

  You can specify DATEFMT values of DATE1 to DATE10, UK (=DATE4), or US (=DATE5), or * (meaning as described previously).

- The need to quote values for characters, hexadecimal (FMT=HEX), hexadecimal numbers (FMT=NUM BASE=HEX) and date format data using the QUOTE operand. By specifying QUOTE=NO (the default) these data types need not be quoted and embedded blanks in NCL variables in free-form text are ignored.

  By specifying QUOTE=YES all of the previously listed data types must be quoted and embedded blanks are treated as real blanks, causing &NDBQUOTE to force-quote all non-null data.

- Tracing of the parsing of free-format text. By default (TRACE=NO), no trace messages are produced.

  By specifying TRACE=YES, a message is produced for each token in the free-form text. This message is subject to the &NDBCTL MSG= setting. The first 20 characters of each token are traced.

- Dumping a scan action table. By default (SCANDEBUG=NO), no dump is produced. By specifying SCANDEBUG=YES, a dump of the generated scan action table is produced. At completion of the scan, a second dump shows record counts.

These options allow you to easily develop applications using NDBs, and to make the procedures robust.

For example, while developing code, use MSG=YES and ERROR=ABORT. This will give useful messages and stop a procedure at the error point. From time to time, TRACE=YES can be used to track obscure syntax errors in free-form text (particularly scan expressions, and problems caused by failing to quote data containing delimiter characters).

When the code is finished, specifying MSG=NO (or MSG=LOG), and ERROR=CONTINUE allows the procedures to handle unexpected errors gracefully.

The DATEFMT= option lets you control the input and formatting of dates. The default behavior (honoring the user or system date format) is normally the best setting. However, when unloading or reloading data, &NDBCTL DATEFMT=NO lets you be independent of the current language settings.

Put It All Together—Unload or Reload an NDB

The following examples illustrate many of the points covered previously. In these examples, a simple pair of procedures is developed to unload and reload logically a database.

These procedures unload any NDB to a VSAM ESDS. The unloading is in RID sequence, unless the database has a sequence key, in which case that key sequences the unloading.

Note: This procedure is described to illustrate some of the features of NDBs. It is not recommended as the best way to unload files. The command NDB UNLOAD unloads an NDB in the same format as the following procedure. The NDB ALTER command can be used to speed up a reload.

Define an Unload File

A file to hold the unloaded data is needed. This example uses a VSAM ESDS. The following definition could be used:

```
DEFINE CLUSTER (NAME(NDB.UNLOAD) -
   NIXD                         -
   RECORSIZE(40 500)            -
   CISZ(4096))
```

This data set must be allocated to your product region:

```
ALLOC DD=UNLOAD DSN=NDB.UNLOAD
```

The data set must be made available to NCL, via the UDBCTL statement. When an empty ESDS is opened, a dummy record is written. Our reload program must skip this record.

```
UDBCTL OPEN=UNLOAD ID=* BUFND=5
```

The unload can now be performed.

Open the Database and Output Unload File

The unload procedure must now open the database, and the destination unload file. Assume that the procedure is invoked from OCS with the name of the NDB and the name of the unload file:

```
$NDBUNLD dbname esdsname
```

Open the database in exclusive mode, to prevent other users updating it while it is being unloaded:

```
&DBNAME = &1  -* save db name to unload
&UNNAME = &2  -* save esds name
&NDBOPEN &DBNAME EXCLUSIVE
&IF &NDBRC NE 0  &THEN &GOTO .OPENERROR -* unable to open
&FILE OPEN &UNNAME
&IF &FILERC NE 8 &THEN &GOTO .OPENERROR  -* cant add recs
```

To preserve any lower-case data, issue &CONTROL NOUCASE:

```
&CONTROL NOUCASE  -* don't fold lower case data
```

Unload Database Level Information

The first step is to write information about the database. You should write a record identifying this as an NDB unload data set, and then write the information returned from an &NDBINFO DB. Use a record type of 01 for the header, and 02 for the database information:

```
&FILE ADD 01 NDB UNLOAD OF &DBNAME &DATE7 &TIME
&NDBINFO &DBNAME DB
&FILE ADD 02 &NDBDBNAME&NDBDBVKL &NDBDBVRL +
         &NDBDBNFLDS&NDBDBNRECS&NDBDBNRID
```

This information will be used by the reload program to verify the key and records lengths of the destination NDB.

Obtain and Unload Field Level Information

To unload and reload the data, you need to know the names of the fields you are unloading, and their attributes. Use &NDBINFO NUMBER= to extract field level information. This information (record type 10) will be written, and vartable built for use when processing data records.

To allow the reload program to easily perform field-level deletion, and so on, you should also write the relative field number as well as the field name when you unload.

Before reading the field information, set up parameters for the sequence that you will define later, defaulting it to RID. If you encounter a sequence key definition, alter the sequence parameter to FIELD=fieldname.

At the end of the field definitions, write a record type 11 to indicate the end:

```
&VARTABLE ALLOC ID=FTAB KEYFMT=NUM DATA=1
&SEQBY = RID
&FNUM = 0
&DOUNTIL &FNUM EQ &NDBDBNFLDS
   &FNUM = &FNUM + 1
   &NDBINFO &DBNAME NUMBER=&FNUM
   &VARTABLE ADD ID=FTAB KEY=FNUM FIELDS=DATA  +
         VARS=NDBFLDNAME
   &IF &NDBFLDKEY = SEQUENCE &THEN &SEQBY =    +
         FIELD=&NDBFLDNAME
   &FILE ADD 10 &FNUM +
      &NDBFLDNAME&NDBFLDFMT&NDBFLDKEY   +
      &NDBFLDNULLF&NDBFLDNULLV&NDBFLDUPD   +
      &NDBFLDCAPS&NDBFLDMAXL&NDBFLDDESC   +
      &NDBFLDUSER1&NDBFLDUSER2&NDBFLDUSER3 +
      &NDBFLDUSER4
   &LOOPCTL 1000
&DOEND
&FILE ADD 11
```

Build a Format for Reading Data

To efficiently read the data, you predefine a format that contains all fields. However, you rename each field to Fnnnn on retrieval. This prevents any clashes with the control fields, if, for example, the database contains a field called I. You also ask that null fields not be returned, to save processing overheads.

You then loop through the vartable built previously, to build a format using START/DATA/END:

```
&I = 1

&NDBFMT &DBNAME DEFINE FORMAT=UFMT START
&NDBFMT &DBNAME DATA FIELDS
&DOWHILE &I LE &FNUM
   &VARTABLE GET ID=FTAB KEY=I FIELDS=DATA VARS=FNAME
   &NDBFMT &DBNAME DATA (F&I = &FNAME NULLFIELD=NORETURN)
   &I = &I + 1
   &LOOPCTL 1000
&DOEND
&NDBFMT &DBNAME END
```

Define the Sequence for Reading

The sequential retrieval path must be defined. The &NDBSEQ verb is used to accomplish this:

```
&NDBSEQ &DBNAME DEFINE SEQ=USEQ &SEQBY
```

The type of sequence, RID or FIELD=sequence key, was set when you read the field definitions.

Unload the Data

You are now in a position to unload all the data in the NDB.

First, write a header record for the data (type 20).

Write out each record as follows:

- A header record, containing the RID and the number of non-null fields in this record. (The RID is only for information, as the reload assigns new RIDs). (Type 21).
- *n* field records, one for each non-null field in the NDB data record (type 22). These records contain the relative field number, field name, and field value.
- A trailer record, indicating the complete NDB data record has been written (type 23). You include the number of non-null fields in the record.

At the end of all data records, a type 29 record indicates the end of the data. This record contains the count of the number of logical records unloaded.

To protect the unloaded data from the current language (and therefore date) settings, issue an &NDBCTL DATEFMT=NO to force dates to be returned in YYMMDD format:

```
&NDBCTL DATEFMT=NO
```

The data unload code is as follows:

```
&FILE ADD 20                        -* header for data portion
&NRECS = 0                          -* num records unloaded
&NDBGET &DBNAME SEQ=USEQ FORMAT=UFMT MODFLD=YES
-* read a record, return the fields modified in ZMODFLD,
-* ZVARCNT
&DOWHILE &NDBRC = 0                 -* till eof
   &NRECS = &NRECS + 1              -* 1 more record
   &FILE ADD 21 &NDBRID &ZVARCNT    -* record header
   &I = 1                           -* field index counter
   &DOWHILE &I LE &ZVARCNT          -* for all defined
                                    -* fields
      &FX = &SUBSTR &ZMODFLD 2      -* get unique field
                                    -* number
      &VARTABLE GET ID=FTAB +       -* and associated
                                    -* field name
         KEY=FX +
         FIELDS=DATA +
         VARS=FNAME
      &FILE ADD 22 &FX &FNAME &FX   -* write field
                                    -* num/name/value
      &I = &I + 1                   -* next field
      &LOOPCTL 1000                 -* prevent blowup
   &DOEND                           -* end all fields
   &FILE ADD 23 &ZVARCNT            -* end record
   &NDBGET &DBNAME SEQ=USEQ FORMAT=UFMT MODFLD=YES
                                    -* get next
      &LOOPCTL 1000                 -* prevent blowup
&DOEND                              -* end record loop
&FILE ADD 29 &NRECS                 -* num logical records
```

This code completes the unloading of data. All that is left is to close the files:

```
&NDBCLOSE &DBNAME
&FILE CLOSE &UNNAME
```

Other people can now use the database. The unloaded data can be processed as necessary, for example, copied to tape.

This example procedure is in the NCL distribution library as member $NDBUNLD. The code is as shown previously. See the source for more information.

Unload Subsets Using Sparse Keys

If an NDB contains disjoint record types, using keys on fields that are not in all records, you can unload just one record type by using one of those keys as the unload sequence. This can be useful for extracting subsets of the database.

Reload an NDB from an Unload File

Having unloaded an NDB, you might want to reload it, possibly to restore the NDB to a previous state, or to take it to another system.

It is assumed that the reload is to a new NDB that has been defined with IDCAMS, allocated to your product region, and an NDB created.

For the purposes of the example, only minimal error checking is performed. The sample procedure can be enhanced in many ways. Some of these are:

- Allow merging of the unload data into an existing NDB
- Display a field selection list and allow field attribute changes, field renaming, and field deletion
- Subset data selection, and so on
- Using LOAD MODE to greatly speed up the load. This requires use of NDB ALTER to build keys.

Open the Database and the Input Unload File

The procedure must first open the database, and the input unload file. It is assumed that the procedure is invoked from OCS with the name of the NDB and the name of the unload file:

```
$NDBRELD dbname esdsname
```

To speed up the reload, start the NDB in DEFER mode. This mode inhibits the database manager from flushing buffers after each &NDBADD, at the expense of an unusable database if the system fails while open this way. To set DEFER mode, the database must have been opened by the UDBCTL command with the options LSR and DEFER. The following NDB command causes deferred I/O:

```
NDB START &DBNAME DEFER
```

Note: The NDB START command can be issued at any time.

Set &CONTROL NOUCASE to protect lower case data:

```
&CONTROL NOUCASE
```

Open the database in exclusive mode, to prevent other users accessing it while it is being reloaded:

```
&DBNAME = &1                         -* save db name to reload
&RLNAME = &2                                  -* save esds name
&NDBOPEN &dbname EXCLUSIVE
&IF &NDBRC NE 0 &THEN &GOTO .OPENERROR    -* unable to open
&FILE OPEN &RLNAME
&IF &FILERC GT 8 &THEN &GOTO .OPENERROR    -* unable to open
```

You must now verify the input file. Read the initial load record (and ignore it), and the second record, which should be the header record:

```
&FILE GET SEQ ARGS               -* should be initld record
&IF &FILERC NE 0 &THEN &GOTO .READERR
&FILE GET SEQ ARGS               -* should be 01 ndb ....
&IF &FILERC NE 0 &THEN &GOTO .READERR
&IF &1.&2.&3.&4.  NE 01.NDB.UNLOAD.OF. +
    &THEN &GOTO .BADFILE
&INDBNAME = &5
&INULDATE = &6
&INULTIME = &7
&WRITE RELOAD FROM BACKUP OF &INDBNAME TAKEN &INULDATE +
      &INULTIME
```

Check Database Attributes

You must now ensure that the destination database can support the reload. Check the following:

- New database VSAM key length is GE unload NDB key length.
- New database is empty of both fields and records.
- New database is freshly created, that is, next RID is 1. (A database emptied by deleting all records, and deleting all field definitions is not suitable).

These tests could be relaxed in a full-function reload.

The code to perform the verification is as follows:

```
&NDBINFO &DBNAME DB                 -* get info about dest
&FILE GET SEQ ARGS                  -* read next record
                                    -* (02)
&IF .&1 NE .02 &THEN &GOTO .READERR -* validate
&IF &NDBDBVKL LT &3 &THEN &DO       -* dest keylen short
   &WRITE destination key length short, load aborted
   &END 16
&DOEND
&IF &NDBDBNFLDS NE 0 &THEN &DO      -* dest db has fields
   &WRITE destination database has fields defined, +
      load aborted
   &END 16
&DOEND
&IF &NDBDBNRECS NE 0 &THEN &DO      -* dest db has records
   &WRITE destination database has records present, +
      load aborted
   &END 16
&DOEND
&IF &NDBDBNRID  NE 0 &THEN &DO      -* dest not just
                                    -* created
   &WRITE destination database not freshly created, +
      load aborted
   &END 16
&DOEND
```

Build Field Definitions

The next records in the unload file are the field definition records. Read them and build field definitions in the new NDB.

Note: Because &10 contains the field maximum length, it is not used.

```
&FILE GET SEQ ARGS                  -* read next record
```

At this point, changes to field attributes could be processed. For example:

```
&IF .&3 = .SURNAME &THEN &5 = UNIQUE
&IF .&3 = .DOB &THEN &5 = YES
```

Also, the relative field number used during unload is available in &2. This can be used during data reloading to alter field values, and so on.

At the end of this step, the new database has all the fields defined.

Load the Data

You can now reload the data records. Each NDB record is represented by one type 21 record, n type 22 records, one for each non-null field, and one type 23 record.

Use &NDBADD START/DATA/END to load the records:

```
&FILE GET SEQ ARGS                  -* should be type 20
&IF .&1 NE .20 &THEN &GOTO .READERR -* bad
&NLOAD = 0                          -* num recs loaded
&OK = YES                           -* flag
&DOUNTIL &OK = NO                   -* rest of DB
   &FILE GET SEQ ARGS               -* get a record
   &IF .&1 = .22 &THEN &DO          -* field (most common)
      -* see note (1)               -* poss changes to
                                    -* data
      &NDBADD &DBNAME DATA +
      &3 = &Q4                      -* add the data
   &DOEND
   &ELSE &IF .&1 = .21 &THEN &DO    -* start of record
      &NDBADD &DBNAME START         -* start add of record
   &DOEND
   &ELSE &IF .&1 = .23 &THEN &DO    -* end record
      &NDBADD &DBNAME END           -* add the record
      &NLOAD = &NLOAD + 1           -* bump num recs
   &DOEND
   &ELSE &OK = NO                   -* otherwise exit
                                    -* until
   &LOOPCTL 1000                    -* prevent blowups
&DOEND
&IF .&1 NE .29 &THEN &GOTO .READERR -* unrecognized record
&IF &2 NE &NLOAD &THEN &GOTO .CNTERR
                                    -* record cnt mismatch
```

Note: Special code, to alter or skip certain fields for example, can be added here.

This completes the reload. All that is left is to close the files:

```
&NDBCLOSE &DBNAME
&FILE CLOSE &RLNAME
```

To flush the database buffers, and protect the reload, re-issue the NDB START with the NODEFER option:

```
NDB START &DBNAME NODEFER
```

This example procedure is in the NCL distribution library as member $NDBRELD. The code is basically as shown previously. See the source for more information.

Chapter 18: Using &NDBSCAN Statements

This section contains the following topics:

Scan Processing

Note: For more information about the steps taken to process an &NDBSCAN statement, see the &NDBSCAN verb description in the *Network Control Language Reference Guide*.

A scan has the following processing steps:

1. The scan request is parsed, and an action table built. The action table is then optimized.
2. The action table is processed and, for each action that can be processed using keys, the key records in the database are used to build intermediate results.
3. If part of the request could not be processed using keys, the final result list (from Step 2) is processed by reading records, and each record is validated against the scan criteria.

   If none of the criteria could be processed using keys, the entire database is scanned.
4. If a sort was requested, the sort keys are extracted from passing records, and an internal sort is performed.

   The final result list is built. Sorting is done during Step 3 or 4 as part of its processing.

These steps seem complex, but no knowledge of the internal processing is necessary to use &NDBSCAN. However, when performance is an issue, use of keyed fields becomes important.

Display the Generated Scan Action Table

The &NDBCTL statement provides an operand that lets you obtain a display of the generated scan action table, both before the scan is processed, and after. This scan debug information can be quite useful for determining why a scan request does not return the expected records.

To obtain this display, code the following statement prior to the &NDBSCAN request:

```
&NDBCTL SCANDEBUG=YES
```

```
&NDBCTL MSG=YES or MSG=LOG must be in effect, otherwise no messages are returned.
```

The first part of the displayed table represents the parsed scan request. Each relation in the request, for example, X = Y, generates a line in the table. All generic or range field names are expanded to the full list, and actions to combine previous results, using AND, OR, or NOT, are shown. The text of the &NDBSCAN request is also displayed.

The second part of the table, produced after the scan completes, shows how many records passed each phase, and whether a scan of the actual data was required.

Process Scan Results

An &NDBSCAN statement can generate a list of the records that pass the supplied criteria. The list is optional, and, if it is not needed, just the number of records and the RID of the first passing record can be returned.

This list is treated like a sequence, defined by an &NDBSEQ statement. Records can be read using the &NDBGET statement, using SEQUENCE=name, where name is the same name as specified on the &NDBSCAN SEQUENCE=name.

The list can be read forward or backward, and records can be skipped.

The order of records returned is undefined, unless SORT=expression was specified on the &NDBSCAN statement. This is to allow the scan request to be optimized. If the scan was sorted, the records are ordered based on the nominated sort fields, when retrieving FWD.

A sorted scan list that has exactly one full-field sort key, can be repositioned by &NDBSEQ RESET REPOS, and, as an extra option available only to sequences built by &NDBSCAN, can be repositioned to the record having a specific RID.

The &NDBSCAN statement lets you specify KEEP=YES, to prevent the scan result list being deleted when an EOF response is returned by &NDBGET.

Differences Between &NDBSCAN Sequences and &NDBSEQ Sequences

There are some important differences between sequences defined by &NDBSEQ, and sequences built by &NDBSCAN:

- &NDBSEQ-defined sequences do not have an in-storage list of records associated with them. Thus, skipping forward and backward automatically takes into account record deletions. For example, if you are reading by RID, and you are positioned on RID 10, a GET FWD SKIP=5 returns RID 15, assuming RIDS 11-14 all exist. If, while positioned on RID 15, RIDs 12 and 14 are deleted, a GET BWD SKIP=5 does not position you on RID 10, but, rather, RID 8 (assuming 8 and 9 exist).

  &NDBSCAN-built sequences are represented as an in-storage list of RIDs (the actual order depending on SORT, and so on). If the list is ordered on RID, the previous example repositions from 10 to 15, and back to 10, as SKIP=n instructs &NDBGET to skip over n-1 entries in the list. Only when the target RID has been deleted, does &NDBGET proceed to the next RID in the list, until a non-deleted record is found.

- &NDBSEQ-defined sequences each take a small, fixed amount of storage to hold information about the sequence.

  &NDBSCAN-built sequences take storage proportional to the number of matching records to hold the list. Thus, you should try to minimize the number of concurrent scan sequences in use, particularly if using &NDBSCAN in an EASINET procedure.

Control &NDBSCAN Resource Usage

Since &NDBSCAN can perform large amounts of I/O, or use large amounts of storage, particularly when sorting, there are four limits that prevent any one scan request from tying up excessive resources. They are:

- Logical VSAM I/O limit
- Working storage limit
- Elapsed time limit
- Passing records limit

These limits are specified using the SYSPARMS command. Each limit can have a default value, to be applied if an &NDBSCAN does not specify an overriding value, and a maximum limit, that is always used to constrain the maximum value which can be coded in any &NDBSCAN.

The &NDBSCAN statement lets you specify overriding values for any of the limits. These values replace the SYSPARM-specified defaults for that scan. If any supplied value exceeds the SYSPARM-specified maximum, the SYSPARM maximum is used.

An &NDBSCAN request that exceeds one of these limits is terminated, and a response code, indicating which limit was exceeded, is returned. The scan debug display indicates the limit values assigned to the scan request.

Scan Expressions

A scan request contains a free form text scan expression. This expression contains:

- Fields - for example, SURNAME, DOB
- Values - for example, SMITH, 04/12/58
- Operators - for example, =, NE
- Connectors - for example, AND, OR

Parentheses can be used to group parts of the expression.

The expression can be spread across several NCL statements, using the START/DATA/END syntax, as per the &NDBADD, &NDBUPD, and &NDBFMT statements. This syntax also allows construction of the scan expression. This is especially useful in table-driven systems, for example, the features table.

The expression consists of a number of scan-tests, connected by AND, OR, NOT, and parentheses. Each test can:

- Test a field, or list of fields (including generic and range field names), against a value, list of values, or generic value(s) or against other fields. This is called a field to field compare.
- The comparison can use the standard relational operators, for example, EQ, NE, GT, and =, >, as well as, special operators:
  - PRESENT-to test for presence
  - ABSENT-to test for absence
  - LIKE-to perform a pattern match
  - CONTAINS-to test, for character fields, for one field containing a value, or other field value
- SQL-like capabilities exist that allow nested scans, where field values from the inner scans are used as input to the tests in the outer scan(s).

The following are examples of valid scan expressions:

```
SURNAME = SMITH AND DOB LT 600101
DESC* CONTAINS MVS

DATECLOSED ABSENT OR DATECLOSED  DATEOPEN PLUS 5
NAME LIKE 'J% SMITH%'
SEX = 'F' AND (STATUS = 'SINGLE' | STATUS = 'DIVORCED')
SEX = 'F' AND STATUS = ANY 'SINGLE', 'DIVORCED'
```

The last two expressions are equivalent. The parentheses are required in the first of these to bind the OR (|) tests together, as AND takes precedence over OR.

Reserved Words

The syntax for a scan expression shows words such as ALL, ANY, FIELDS, and VALUES. These words are used to indicate options in scan tests. Although it might appear that these are reserved words, and thus cannot be used as field names, or unquoted field values, this is not the case.

Note: For more information about the syntax for a scan expression, see the &NDBSCAN verb description in the *Network Control Language Reference Guide.*

There are no reserved words in the syntax for a scan expression.

All keywords are resolved by context. That is, if a certain keyword is allowed at a point in the expression, the presence of that keyword at that point in an expression is regarded as being that keyword. At any other point, that keyword is regarded as a name, value, and so on.

For example, the following are valid scan expressions. Keywords are shown in bold typeface. (Assume that field names and so on are defined.)

```
FIELD ALL = VALUE ANY
ANY FIELDS VALUE, FIELD, ALL CONTAINS ALL VALUES FIELD, + VALUE, ANY
```

This can cause some confusion. Obviously, fields names of FIELD, ALL, and so on, are not recommended.

When generating scan expressions dynamically, it is always a good idea to insert all the optional keywords, to prevent a syntax error when, for example, a search value of FIELDS is provided. For example:

```
&NDBSCAN ...  DATA FIELD1 = &SCHVALUE
```

If the variable &SCHVALUE contained the characters FIELDS, without the quotes, the scan would fail with a syntax error, as the keyword FIELDS is not followed by a field name.

To prevent this, the previous example could be coded:

```
&NDBSCAN ...  DATA FIELD1 = VALUE &SCHVALUE
```

Null Fields

Null (that is, not present) fields are handled in a special way by &NDBSCAN:

- A null field never matches a field to value test. A record with field SURNAME not present is not equal to a supplied value, nor is it not equal to a supplied value. This 'null result' carries through the scan action table.
- A null field can only be selected using the ABSENT or IS NULL operators.

This handling of null fields by &NDBSCAN is consistent with other &NDB verbs. Thus, disjoint record types work as expected.

Field to Field Comparisons

You can compare one field with another in a record. A good example of this is finding all records with WITHDRAWALS GT BALANCE.

Note: Field to field comparisons involve scanning records. Always try to have other (keyed) criteria ANDed with these criteria.

When performing field to field comparisons on numeric, float or date fields, an adjustment amount (taken as a number of days, for date format data), can be specified. For example, the following statement matches all records where the field DATECLOSED was greater than the field DATEOPEN plus 10 days:

```
DATECLOSED GT DATEOPENED PLUS 10
```

The amount must be an integer; floating point numbers are not supported.

When performing nested scans (sub-selects), you can perform a field-to-field comparison when one of the fields is the current value in an outer scan. This is called a correlated query.

Use &NDBQUOTE to Protect Special Characters

The &NDBQUOTE built-in function should be used to protect data values whenever there is the possibility of delimiter characters (for example, =, &) being present. This recommendation also applies to arguments supplied in a scan expression.

For example, to search for a value of A=B, use these statements:

```
&SCHARG = &NDBQUOTE A=B
&NDBSCAN MYNDB SEQ=S1 DATA FIELD1 = &SCHARG
```

Failure to quote the value results in a syntax error because the equal sign is treated as a delimiter.

Search for Lowercase Data

An NDB can store character data in lowercase. When a character field is defined, CAPS=YES is assumed, which means that both the stored data and the key are folded to uppercase. Two other options are available:

CAPS=NO

The data is left as is, that is, lowercase data is left lowercase, including in the key. For example, ABC and abc are regarded as different values.

CAPS=SEARCH

The data is left as is, that is, lowercase data is left lowercase. If the data is keyed, the key is folded to uppercase. Thus, the values ABC and abc are regarded the same when building a key, but, when data is returned, the lowercase version is retained.

For &NDBSCAN, CAPS=SEARCH also applies to non-keyed fields for processing. That is, when reading data records, fields defined with CAPS=SEARCH are folded to uppercase when they are examined.

Supplied search arguments are retained in lowercase (or as entered), and are upper cased as required (that is, when comparing to CAPS=YES or CAPS=SEARCH fields).

Note: &CONTROL NOUCASE must be in effect to preserve lowercase information supplied on an &NDBSCAN statement.

CONTAINS

The CONTAINS operator is extremely useful when scanning text in a database. When combined with generic or range field names, a single expression can perform quite sophisticated lookups.

Some notes about the use of CONTAINS:

- CONTAINS involves scanning records. Always try to have other criteria ANDed with CONTAINS, to reduce the number of records that must be scanned.
- The field(s) on the left of CONTAINS are padded at each end with one blank, for the purposes of searching. This allows words to be searched for, by providing blanks around your search arguments, without worrying that a word at the front or back of a field will not be matched. For example, a field value of A SENTENCE OF WORDS is regarded as A SENTENCE OF WORDS when processed by CONTAINS. Thus, searching for the value A will succeed, even though the data does not contain a blank in front of the A.
- The ANY and ALL keywords allow some requirements for text to be adjacent to be tested. For example the following statement only matches records where any one of the fields prefixed by DESC contains both WORD1 and WORD2:

  ```
  ANY FIELD DESC* CONTAINS ALL WORD1, WORD2.
  ```

  If each DESC... field held a sentence, this is equivalent to asking for a sentence containing both words.

  Fields in the same record can contain the search argument(s). Thus, it is possible to find all records where a given character field in the records is contained within another character field.

LIKE

The LIKE operator allows pattern matching to be performed. The NOT LIKE operation does the same, but matches records without the pattern.

The argument string for LIKE can contain any characters except for two special characters: _ (underscore) and % (percent). These characters only match themselves (the CAPS=NO/SEARCH rules are honored).

The special characters obey the following rules:

- Underscore (_) matches any one character, but there must be a character in the data at this position.
- Percent (%) matches zero or more characters.

These special characters can be used as many times as required in a search string. Some examples follow:

ABC

Matches only records with ABC in the field.

ABC%

Matches on values starting with ABC

ABC_%

As above, but at least one character must follow ABC

%XYZ

Matches fields ending in XYZ

%mmm%

Matches fields with mmm somewhere in them

%FRED%_%BLOGGS%

Matches a field containing the strings FRED and BLOGGS in that order, with at least one character between them

_ _ _

Matches a field exactly 3 characters long

_ _XY_%

Matches a field at least 5 characters long, with XY in columns 3 and 4

It is evident that LIKE is a powerful operator and consequently can use significant CPU resources.

The first two examples in this table can also be done using keys. LIKE attempts to use keys to fully or partially (whenever a LIKE argument has non-special leading characters) determine matching records.

Use the Results of a Previous &NDBSCAN

The syntax, SEQUENCE seqname allows a scan to use as part of its input, the result list from a previous scan. This avoids having to re-evaluate an entire previous scan, just to add some more criteria to it.

In interactive applications, this can be very powerful. For example, after performing a scan based on user input, a panel can be displayed informing the user of the number of hits. One option the user can have is to add extra criteria, and see the result of the extra criteria, as applied to the current result list. The display, extra criteria, and so on, cycle could be repeated. By using the previous scan as input, large amounts of system resources can be saved. Also, as the user ascends the nested levels of display, previous results are still valid.

This option is also useful when a scan expression that is too complex to handle in one statement is needed. It can be broken into parts, and the results combined.

For example:

```
&NDBSCAN MYNDB SEQ=S1 DATA SURNAME = 'SMITH'
&NDBSCAN MYNDB SEQ=S2 DATA DOB LT 600101
&NDBSCAN MYNDB SEQ=RESULTS DATA SEQUENCE S1 AND +
    SEQUENCE S2
```

A scan expression can just include a previous scan result. This is useful when you want to re-sort a scan result without rebuilding as list. For example:

```
&NDBSCAN DB2 SEQ=S1 SORT=FIELD1 DATA ...scan expr
&NDBSCAN DB2 SEQ=S2 SORT=FIELD2 DATA SEQUENCE S1
```

SQL-like Operators

Scans can perform some SQL-like functions:

- The IS [NOT] NULL operators provide equivalent functionality to the PRESENT and ABSENT operators, but are SQL-compatible.
- The [NOT] BETWEEN operators are equivalent to the =value:value and p=value:value operators, and are SQL-compatible.
- The [NOT} IN operators are equivalent to the [p]= value-list operators, and are SQL-compatible.
- The SELECT clause lets you perform sub-selects, where a list of values derived from a set of records matching some criteria can be used as input to an outer-level scan.

The EXISTS operator uses the fact that at least one record exists in the inner select/scan to determine truth or falsity.

A correlated select is possible. This causes an inner scan to be re-executed once for each other outer record that passed part of a scan. The outer record field values are used in the inner scan as comparison values.

Efficient Use of &NDBSCAN

&NDBSCAN can be misused. The scan limits discussed previously are designed to prevent excessive use of resources. When designing applications that use &NDBSCAN, the following should be taken into account:

- Although you can search on any field, use of keyed fields, connected by AND, at the outer level of the scan expression, greatly reduces I/O and elapsed time. Even one keyed field can have a significant impact. For example, the following statement reads every record on the database, if field NAME has no key:

  ```
  NAME = 'SMITH' AND ADDRESS CONTAINS 'STREET'
  ```

  PRESENT and ABSENT use keys if possible. PRESENT makes better use of keys. LIKE uses keys if there are non-special ('_', '%') characters at the front of the pattern. It might still need to examine the actual record.

  The CONTAINS operator always requires records to be read to evaluate success or failure. If you are performing many keyword searches of text using CONTAINS, consider storing the words of the text field as individual, keyed fields. These can then be searched for directly. For example the following statement scans the entire database:

  ```
  DESCRIPTION CONTAINS 'WORD'
  ```

  If, however, as well as field DESCRIPTION, you had fields DESCWD01-DESCWD10, all keyed, containing the individual words of the field DESCRIPTION (&PARSE could be used), you could code the following statement which could use keys:

  ```
  ANY DESCWD* = 'WORD'
  ```

- For interactive applications, encourage the use of scan criteria which reduce the number of hits. If a given scan returns more hits than could be validly processed by the user, display a message requesting more and/or more selective criteria. The scan option to use the results of a previous scan can also be useful in this case.

- Avoid the use of OR at the highest level of the scan expression, if either side of the OR involves a non-keyed field, or PRESENT, ABSENT, or CONTAINS-a full database scan results.

- Only use SORT when absolutely necessary. Sorting will normally involve reading all the records that pass the criteria, extracting the sort field, performing an in-storage sort, and then building the result list. The sort key extraction is done concurrently with final record scanning if non-keyed, and so on, criteria are to be processed.

  This processing can consume large amounts of I/O, as well as storage for the sort keys, and can take significant time.

- Nested scans (SELECT causes) can use large amounts of storage to store inner results.

  Correlated scans can perform multiple passes over the database. This is because an inner scan can be called once for every record in the NDB, and it too can read the NDB, or a large portion of it.

- If you only want to know whether any records at all have, or have not, passed the scan criteria, and you are not interested in the exact number or specific IDs of any such records, then use parameter RECLIMIT=1. &NDBSCAN RECLIMIT=1 does not set the &NDBRID variable.
- If sorting is unavoidable, for example, because you need to be able to reposition in the output sequence, always try to minimize the number of records passing the scan.

Chapter 19: Using Advanced Program-to-Program Communication

This section contains the following topics:

Advanced Program-to-Program Communication (APPC)

The SNA APPC is a high-level application programming interface that allows program-to-program communications and the development of distributed applications between network nodes that support Logical Unit type 6.2 (LU6.2). LU type 6.2 communication facilities form the basis for SAA's Common Programming Interface for Communications (CPI-C).

In this implementation of APPC, the high-level application programming interface is provided by the NCL &APPC verb which provides access to a full set of LU6.2, or APPC, programming capabilities.

These facilities let an NCL procedure communicate with another APPC application in a structured manner as defined by the LU6.2 protocol. The partner application can exist within the same, or a remote system, or any other application system that supports LU6.2 protocols.

In addition, this implementation of APPC supports a number of extensions that assist the development of client/server applications within NCL or spanning other APPC platforms.

Before writing NCL procedures that use these APPC facilities, you should familiarize yourself with the fundamental APPC concepts and implementation procedures. For more information, see the *Reference Guide*. In addition, see IBM's Communications Server SNA Programmer's LU 6.2 Reference, which is the authoritative source for detailing the LU6.2 verb set.

APPC Conversations

NCL procedures (and transaction programs in general) communicate by establishing a communication path called a conversation. Conversations use LU6.2 sessions to exchange data and protocols between the communicating procedures. All conversations consist of three main phases:

- Conversation initiation (allocation and attach processing)
- Data exchange (sending and receiving data operations)
- Conversation termination (deallocation processing)

Once a conversation is allocated to a session, a send-receive relationship is established between the participating programs. Initially, the procedure that requested the conversation is allowed to issue send data verbs while the other procedure issues receive data verbs. This send-receive relationship can change many times during the life of the conversation.

To terminate the conversation the procedures request deallocation processing, by issuing the appropriate deallocate verb. The following sections give a detailed description of the conversation phases and the associated verbs.

Note: This implementation of APPC enables NCL procedures on the same system to communicate without the need to use SNA sessions. This is a highly effective means of data transfer between NCL processes.

LU6.2 Verb Set

The LU6.2 verb set defines a structured means of communication between two programs. A strict protocol exists at many layers in the LU6.2 implementation, including the definition of the application verb set and conversation states. Most APPC requests are handled by the NCL &APPC verb, while conversation state and other status indicators are accessible to NCL through a range of system variables.

&APPC Verb

All actions on a conversation are supported through the &APPC verb. The specific request is identified by the keyword immediately following the &APPC verb. In general, the keyword identifying each request corresponds closely to the LU6.2 architected verb syntax such that the specific LU6.2 architected verb is apparent from the NCL syntax. The relationship between the NCL options and the LU6.2 architected verb set is described in the reference section for the particular &APPC request. The set of &APPC requests is as follows:

- &APPC ALLOCATE_DELAYED
- &APPC ALLOCATE_IMMEDIATE
- &APPC ALLOCATE_NOTIFY
- &APPC ALLOCATE_SESSION
- &APPC CONFIRM
- &APPC CONFIRMED
- &APPC DEALLOCATE
- &APPC FLUSH
- &APPC PREPARE_TO_RECEIVE
- &APPC RECEIVE_AND_WAIT
- &APPC RECEIVE_IMMEDIATE
- &APPC RECEIVE_NOTIFY
- &APPC REQUEST_TO_SEND
- &APPC SEND_DATA
- &APPC SEND_ERROR
- &APPC TEST

Note: Your product does not support LU6.2 sync-point processing, and hence the verbs and states that implement this are unsupported.

Conversation States

Conversations are managed by transition through a number of states. The state of a conversation is reflected in the &ZAPPCSTA system variable, and can be one of the following:

- RESET
- SEND
- RECEIVE
- DEFER_RECEIVE
- DEFER_DEALLOCATE
- CONFIRM
- CONFIRM_SEND
- CONFIRM_DEALLOCATE
- DEALLOCATE

Conversation Processing

Only a small number of verbs are acceptable to NCL from any given conversation state, otherwise a state error ensues. Following the successful completion of each verb, either the state remains unchanged, or a single state transition is possible. This means that the programmer can always determine the next course of action for the conversation. If verb completion is not successful then further analysis of the reason, by examination of conversation status fields, might be required to determine the most appropriate course of action to follow.

In the simplest analysis, the APPC protocol is a flip-flop communication channel where, in normal operation, the procedure currently sending controls the data flow. Hence, the first speaker procedure continues sending data until it decides to stop. It might decide to stop only when all data has been sent. For example, after sending a request for a database query, to reverse the direction of data flow and receive the response. Or it might pause during transmission to request confirmation (actual receipt by the partner procedure) of that portion of the data sent so far.

Only when the current speaker changes direction can the partner procedure send data the other way. However, if some condition arises that is generally unrecoverable, an error indication can be sent to the current sender to cause transmission to cease, and allow the current receiver to enter send state.

Within your product, all information being sent is buffered until enough data is accrued to warrant transmission, or the need for some application response dictates that an actual network transmission take place. This provides for efficient use of network resources, but means that to synchronize local and remote processing, the application must manage the protocol appropriately. This is achieved through use of the confirmation protocols.

Return Codes and System Variables

Following each verb the &RETCODE and &ZFDBK system variables are set to indicate the success or otherwise of the &APPC request.

In addition, for each conversation, a set of system variables is maintained that provides information concerning the conversation status:

&ZAPPCELM

Message from an error log GDS variable

&ZAPPCELP

Product set information from an error log GDS

&ZAPPCID

Conversation identifier

&ZAPPCIDA

The conversation identifier for the transaction that started the NCL process

&ZAPPCLNK

Local link name

&ZAPPCMOD

Session mode name

&ZAPPCQLN

Network qualified local luname

&ZAPPCQRN

Network qualified remote luname

&ZAPPCRM

Current receive map name

&ZAPPCRPI

Received protocol indicators

&ZAPPCSM

Current send map name

&ZAPPCSTA

Conversation state

&ZAPPCSYN

Conversation sync_level

&ZAPPCTYP

Conversation type

&ZAPPCWR

What_received indicator

&ZAPPCWRI

What_received short indicator

&ZAPPCRTS

Request_to_send indicator

&ZAPPCTRN

Transaction identifier

Conversation Allocation

An executing procedure establishes a communication path, termed a conversation, by the process known as allocation. An allocation request is deemed to take place from reset state, and can be issued as one of the following verb options:

- &APPC ALLOCATE_SESSION (or just &APPC ALLOCATE)
- &APPC ALLOCATE_DELAYED
- &APPC ALLOCATE_IMMEDIATE
- &APPC ALLOCATE_NOTIFY

Transaction Identifier

One of the parameters supplied on the allocate verb is the TRANSID (transaction identifier) which directs your product to the APPC Transaction Control Table, or TCT. An appropriate TCT entry for the transaction must be located or the conversation fails. The TCT entry is used to verify the request before further information is extracted to complete the request if necessary.

Destination Selection

The TCT entry can contain default destination information, by way of a link name or network LU name, to be used for the request. However the LINK or LUNAME parameters on the allocate verb can be used to override any TCT destination information (and will be required if the TCT has no default destination information).

Once a destination has been isolated resources are allocated to set up the communication path, and eventually a procedure or program is attached in the target system to service the request.

If the target system is the local system, an NCL procedure is started as an attached procedure. Such procedures are generally written to perform a specific function, or possibly a range of functions, on behalf of the invoking procedure.

If the target system is outside of the local system, an SNA LU6.2 session connecting the remote destination must be located. This session is then allocated to the conversation for the conversation duration. The fact that a session is interposed between the communicating procedures is completely transparent to the allocating procedure.

Allocation and Sessions

To complete an allocation to a remote destination a session is required. If no session to that destination currently exists, then for all requests except &APPC ALLOCATE_IMMEDIATE, an attempt will be made to establish an APPC link to the remote LU. Other than the fact that it is an automatic link activation request, the result is no different from an operator starting a link by command.

For the &APPC ALLOCATE_SESSION and ALLOCATE_NOTIFY requests, the verb will not complete until a session can be assigned for the conversation. This can involve waiting for new sessions to be established, or waiting for active conversations to end and free up a session.

For the &APPC ALLOCATE_DELAYED request, a delayed session assignment is allowed. The verb completes as though a session can be assigned and processing continues. However, if transmission is actually required at some stage, the procedure will be suspended until session assignment can be performed.

The &APPC ALLOCATE_IMMEDIATE request only completes successfully if a session can be guaranteed immediately for transmission.

Set Program Initialization Parameters

User data can be passed as parameters on the allocate verb. These can be in the form of one or more NCL tokens. Each NCL token passed as a parameter on the allocation will appear as a separate parameter in the remote end. These parameters are available to the attached procedure as program initialization parameters, described in the next section.

Allocation Completion

After a successful allocation, the procedure is placed in send state and the &ZAPPCID system variable is set to provide the identifier of the conversation created. More than one conversation can be allocated and operated concurrently by a single NCL procedure.

Note: The allocation can complete without any transmission taking place, and hence the procedure enters send state before the target procedure is invoked to service the request.

Attach a Procedure

At some time following allocation, data is actually transmitted to the target system in the form of an attach request.

When an attach request is received by your product, the TCT is examined to locate the procedure that will service the request. The procedure is said to be attached and a copy of the procedure is invoked as a new NCL process.

Client/Server Terminology

Client/server processing lends itself well to the use of APPC as a protocol. For this reason, the term client is often used to mean the allocating procedure, and server to mean the attached procedure. Many of the &APPC verb extensions are designed to support the client/server model.

Execution Environment

If the conversation is executed as an unsecured transaction, the procedure will execute in the background server (BSVR) environment. Otherwise an APPC user region is located (or created and signed on if none exists), for the user ID carried in the attach request, and the procedure started within that user region. When no further NCL activity exists in an APPC user region it is signed off and deleted.

Access Program Initialization Parameters

Once attached, the NCL procedure begins execution as any other NCL procedure, except that it already has an active conversation.

If program initialization parameters were passed on the allocation request, they are accessible as the standard NCL parameters &1 &2 &3through to &n in the usual manner. Each token represents a single passed parameter and hence the data available is limited to the current maximum token size.

Attach Processing

The conversation that attached the procedure is identified by the &ZAPPCIDA system variable and is also the current conversation identified by the &ZAPPCID system variable. The procedure is in receive state on this conversation. An attached procedure has access to all the usual NCL facilities and can allocate further conversations if desired.

Send Operations

Data can be sent on a conversation only from send state. Send state is entered in one of the following ways:

- Automatically following a successful allocation
- At any time when the remote conversation partner that is currently sending decides to stop sending and enter receive state, thereby placing the local end in send state.
- After issuing a SEND_ERROR which forces send state locally

Send Data

The only verb used to send data is:

```
&APPC SEND_DATA
```

Data can be in the form of one or more NCL tokens, or it can be a Mapped Data Object, or MDO. Even where multiple tokens are identified on the send operation, they are packaged as a single unit of transmission, called a GDS variable, to the remote end where they will eventually become available from a single receive request.

There is no limit to the number of variables or MDOs, or total size of the GDS variable used for transmission.

Following a successful send data operation the conversation remains in send state. Otherwise the return code information should be examined for details on the send failure.

Data Mapping Support

APPC can send a map name of up to 64 characters as control information when data is sent. This map name is used by the remote system to interpret the contents of the data transmitted and indicates how it should build any local data structures.

Data Mapping for NCL Tokens

When sending NCL tokens, a map name of $NCL is sent by default with the data to indicate that the data is comprised of one or more NCL tokens. If the receiving system understands the $NCL structure, it can decompose the datastream into the individual tokens sent. In this way NCL allows a single send to specify multiple tokens as a range or list, and in the target system a single receive can re-create such a range or list. For example, the request:

```
&APPC SEND_DATA VARS=A* RANGE=(1,10)
```

can be sent, and satisfies a remote request:

```
&APPC RECEIVE_AND_WAIT VARS=B*
```

such that the variables B1,B2,...,B10 in the remote system are created with the same values as A1,A2,...,A10 in the sending system.

You can override the map name sent. However, if a map name other than $NCL is specified the tokens are not structured as above for output, but are simply concatenated together to form the single GDS variable which is the unit of transmission. When communicating with a non-NCL system, this may let you construct the appropriate pieces of the data in separate tokens before sending them as a single datastream. The interpretation of the block of data received in the remote system is then implied by the map name sent.

Data Mapping and Mapping Services

When sending data from an MDO, the MDO name specified determines the length of the data unit to be sent. The map name sent by default is the fully qualified map name hierarchy as defined to Mapping Services.

For example, if a CNM alert record was in an MDO named NEWS, mapped by the map named CNM, then the second Product Set Identifier sub-vector within that record might be referenced from NCL as:

```
NEWS.NMVT.ALERT.PSID{2}
```

If the Product Identifier sub-vector within it was the subject of a send operation it can be sent by:

```
&APPC SEND_DATA MDO=NEWS.NMVT.ALERT.PSID{2}.PRODID
```

However, the fully qualified map name hierarchy for the object is:

```
CNM.NMVT.ALERT.PSID.PRODID
```

as defined to Mapping Services, and this will be the map name sent. If the target system understands this mapping hierarchy it will reconstruct the MDO and map it accordingly. For example a receive request of:

```
&APPC RECEIVE_AND_WAIT MDO=NEWSREC
```

would result in the object being:

```
NEWSREC.NMVT.ALERT.PSID.PRODID
```

Note: All name instances are now implied to be 1, as only the second PSID instance was selected for the send. No data existing in any of the elements of the intervening name segments is sent, however the qualified names are sent to provide the context of the actual data transmitted.

If the entire MDO had been sent, for example:

&APPC SEND_DATA MDO=NEWS

then both PSIDs would be received and the entire NEWSREC structure received would be identical to the NEWS structure sent.

Again, you can choose to override the map name sent, allowing the target system to implement data interpretation independently.

Send Data When Data Mapping Is Not Supported

Not all LU6.2 systems support data mapping. If data mapping is not supported (as defined in the Option Set Control Table selected for the destination system) then no map names are sent and all data sent is always transmitted as is.

If a number of tokens are specified on the SEND_DATA request then they are simply concatenated together to form the single unit of data transmission. If an MDO is sent then the data from the MDO comprises the entire data unit. The structure of this data unit, and its interpretation in the remote system are the responsibility of the application developer.

Request Confirmation of Data Sent

While in send state a procedure can request confirmation of any data sent. This is achieved by the verb:

```
&APPC CONFIRM
```

Processing is suspended until the remote procedure has received all data sent up until that point and receives the confirmation request. The normal response is:

```
&APPC CONFIRMED
```

This response indicates that all data was received and processed normally. The receipt of the confirmed response satisfies the confirm request and allows the sending procedure to continue processing in send state.

Force Data Transmission

If deemed necessary by the procedure it can, while in send state, force the transmission of any data buffered by APPC with the request:

```
&APPC FLUSH
```

All queued data is scheduled for transmission by this request, however it is usually unnecessary as data will be transmitted as it accumulates during normal send operations. When a reasonable amount of data is buffered, or if requests which require some response (such as an &APPC CONFIRM) are issued, then a transmission is scheduled automatically by the system. A flush operation does not alter the state.

Switch State from Send to Receive

When a procedure has completed all send operations and expects some response data from the conversation partner it can switch from send to receive state by issuing:

```
&APPC PREPARE_TO_RECEIVE
&APPC PREPARE_TO_RECEIVE TYPE=FLUSH
&APPC PREPARE_TO_RECEIVE TYPE=CONFIRM
```

If no type is specified then it will default to FLUSH if the conversation has a sync_level of NONE, or CONFIRM if the conversation has a sync_level of CONFIRM (as set by the allocation request and reflected in the &ZAPPCSYN system variable).

The FLUSH option simply ensures all data is sent to the other end with a request to change direction. For the CONFIRM option the remote procedure must issue an &APPC CONFIRMED request before the verb completes and receive state is entered.

If data is expected in response then a confirmation can usually be avoided by simply issuing, from send state, either request:

```
&APPC RECEIVE_AND_WAIT
&APPC RECEIVE_NOTIFY
```

These requests will perform an implied &APPC PREPARE_TO_ RECEIVE TYPE=FLUSH before placing the procedure in receive state.

Receive Operations

Data can be received on a conversation only while in receive state. Receive state is entered in one of the following ways:

- Automatically when a procedure is attached by an allocation request
- At any time when the procedure decides to stop sending and enter receive state voluntarily
- When an error is received from the remote system, forcing the local procedure into receive state

Receive Data

The verb options available for requesting received data are:

```
&APPC RECEIVE_AND_WAIT
&APPC RECEIVE_IMMEDIATE
```

while an asynchronous notification that data has been received (without actually receiving it into target variables) can be requested by:

```
&APPC RECEIVE_NOTIFY
```

Following notification, the RECEIVE_IMMEDIATE request can be used to retrieve the data. When data is received, it can be placed in one or more tokens, or into an MDO.

Following a receive operation the &ZAPPCWR and &ZAPPCWRI indicators are set explaining the nature of the data received. It is possible to get a successful receive (&RETCODE 0) when no data is placed into the target variables. However the what-received indicator could show that some request, such as a confirm, has been received and hence the procedure should issue the &APPC CONFIRMED response.

Receive Data into NCL Tokens

Either a list or range of NCL tokens can be specified as the target variables for a receive operation. For example:

```
&APPC RECEIVE_AND_WAIT VARS=$* RANGE=(1,20)
&APPC RECEIVE_AND_WAIT VARS=(A,B,C,D)
```

The contents of each variable will be set depending upon the data mapping for the conversation. If no mapping is supported, or if the map name is other than $NCL, then the received GDS variable (which is the unit transmitted) is segmented according to the maximum token size by default. You can specify individual segment sizes when using the list form by specifying a parenthesized integer after each variable name, for example:

```
&APPC RECEIVE_AND_WAIT VARS=(A(10),B(8),C(4),D(32))
```

If mapping is supported and the map name is $NCL, then the received GDS variable, which is the unit transmitted, is assumed to be correctly formatted by the sending NCL system. Its structure maintains the contents of the individual tokens that were specified on the send operation, and the target variables are reconstructed identically.

In any case, no data transformation takes place. If the transmitted data contains characters that cannot be displayed, subsequent processing might require its conversion to hexadecimal characters (through the &HEXEXP built-in function). This is your responsibility.

If insufficient variables are supplied to receive all the data transmitted in the data unit, the residual data is lost.

Note: The received map name is in the &ZAPPCRM system variable.

Receive Data into an MDO

As an alternative to the use of NCL tokens an MDO can be specified as the target for a receive operation, for example:

```
&APPC RECEIVE_AND_WAIT MDO=APPCREC
```

The MDO is structured depending upon the data mapping for the conversation. If no mapping is supported, or if the map name is unknown to Mapping Services, then the data from the received GDS variable which is the unit transmitted forms the entire contents of the MDO. The MDO is unmapped, but the MDO name refers to the entire data transmitted.

If mapping is supported and the map name is known, then the data from the received GDS variable which is the unit transmitted, is assigned into the MDO, specified according to the map name received. If the map name consisted of more than one name segment, then the first name segment is assumed to be the actual map name, and the remaining segments qualify the data in the usual Mapping Services manner.

For example, if the map name received was:

```
CORPORATE.PAYROLL.EMPLOYEE
```

and the receive specified was:

```
&APPC RECEIVE_AND_WAIT MDO=USER
```

then the MDO structure named USER.PAYROLL.EMPLOYEE and mapped by the Mapping Services map name CORPORATE, is assigned the received data. However if the map name CORPORATE was unknown to Mapping Services then the MDO structure named USER contains all the data and it is unmapped.

You can determine the received map name through the &ZAPPCRM system variable and process the MDO contents accordingly. For example, if a transmitted map name is unknown to Mapping Services, or is an invalid Mapping Services map name, the data can be placed into an unmapped MDO. The map name is then examined and the MDO assigned to a new structure with Mapping Services mapping through the &ASSIGN verb.

Respond to a Confirmation Request

Following a receive operation, the &ZAPPCWR might indicate that a confirmed response is required if the &ZAPPCWR value is:

```
CONFIRM
CONFIRM_SEND
CONFIRM_DEALLOCATE
```

in which case the procedure can respond:

```
&APPC CONFIRMED
```

after which receive state, send state, or deallocate state is entered respectively.

Receive a Send Indication

Following a receive operation, the &ZAPPCWR can indicate that the local procedure has entered send state if the &ZAPPCWR value is SEND. This indicates that the remote end has completed its send operations and has entered receive state.

Receive a Deallocation Indication

Following a receive operation, the &ZAPPCWR can indicate that the conversation has been terminated unconditionally by the remote procedure. In this case &RETCODE 4 is returned, and the &ZAPPCWR value is DEALLOCATE, the only allowable action is to issue:

```
&APPC DEALLOCATE TYPE=LOCAL
```

after which all conversation information is removed from the system.

Error Processing

Either conversation partner can send an error indication at any time by issuing &APPC SEND_ERROR.

When issued from send state, the remote end (in receive state) will get a return code indicating program_error_no_truncation, and &RETCODE is 8. This indicates that the error sent did not cause any loss of data. No state changes occur.

When issued from receive state, the remote end (in send or receive state) will get a return code indicating program_error_purging, and &RETCODE is 8. This indicates that the receiver is purging all subsequent data sent and the sender has an obligation to enter receive state (if it has not already done so).

The effect of a SEND_ERROR is to halt communication and place the error sender in send state. However it should be used with discretion as it is most disruptive of normal data flows. Usually a reason for the error will follow as a normal data send operation.

Some errors can be detected by the APPC Services layers, and will appear as svc_error_purging and so on. For example, if a procedure which is one end of a conversation terminates abnormally, a deallocate abend condition is raised and an error notification is sent to the other partner. A message accompanies such a condition and will appear in the log of the remote system. It can be accessed through the &ZAPPCELM system variable before the &APPC DEALLOCATE TYPE=LOCAL statement is issued.

Conversation Deallocation

Regardless of which end started a conversation, any procedure can terminate it from send state by issuing one of the deallocate requests:

```
&APPC DEALLOCATE
&APPC DEALLOCATE TYPE=FLUSH
&APPC DEALLOCATE TYPE=CONFIRM
&APPC DEALLOCATE TYPE=ABEND
```

If no type is specified, then it will default to FLUSH if the conversation has a sync_level of NONE, or CONFIRM if the conversation has a sync_level of CONFIRM (as set by the allocation request and reflected in the &ZAPPCSYN system variable).

A TYPE=CONFIRM deallocation is conditional upon the remote procedure issuing an CONFIRMED response. The only other valid response is a SEND_ERROR, which means the conversation remains active and the error sender is placed in send state. If &RETCODE of 0 is returned the deallocation was successful.

A TYPE=FLUSH or TYPE=ABEND is unconditional as long as the request is accepted. A &RETCODE of 0 indicates that the conversation is terminated.

Sample Conversations

A set of simple APPC conversations illustrating the use of the &APPC verb is provided in the distributed sample library.

More information:

Run the Sample APPC Conversations (see page 494)

&APPC Return Code Information

All &APPC verb options complete by setting a number of NCL variables with return code information.

General completion information is contained within the &RETCODE system variable and qualified by the &ZFDBK system variable. &RETCODE values of 0 and 4 occur in normal operation, while &RETCODE values of 8 or higher indicate an error condition. If an error is detected, the &SYSMSG user variable is set providing an explanation of the error condition.

A number of APPC system variables are available providing information about the current conversation being operated. For example, following a receive operation the &ZAPPCWR and &ZAPPCWRI (what-received indicators) provide information about what satisfied the operation.

Note: For more information about &RETCODE and &ZFDBK system variable settings, see the &APPC verb description in the *Network Control Language Reference Guide*.

Application Design

An important aspect of APPC programming is the consideration of application design. The protocol and verb set provided by APPC does not dictate a specific mode of use and can be flexibly adapted to a number of situations. In this sense, the application's use of the APPC verb set should be considered as part of the overall application design, and this further implies that the application spans the conversation.

Thus what is ostensibly a single application can be split into two (or more) procedures which communicate using the NCL &APPC verb. This form of application lends itself well to client-server type roles. Commonly, the client procedure will act as the man-machine interface, providing the presentation aspects of the application.

The server procedure can deal with the data organizational aspects of the application, fetching and returning to the presentation procedure one or more records depending upon the request.

There are some important advantages in this approach:

- The code for handling the functional or presentation aspects is physically separate from the code for dealing with data organization aspects. This simplifies the processing required in each part, and assists future maintenance. For example, the server procedure could be completely replaced at any stage to handle a new database or file system.
- As the code is separate, the procedures can be executed in different systems without any change to the application. For example, the client end can always reside in a work station while the server end is in a host system. However, they can also execute within the same system. An additional advantage is that should a server procedure need to be moved to another system, for example if a database has been moved, then no code changes are necessary. Simple changes to the APPC Control Tables can be used to redirect transactions to the new system.
- Procedures are executed on demand. Rather than initializing a process to wait for work, the required procedure can be attached when a demand for its services is received.

More information:

APPC Client/Server Processing (see page 384)

Chapter 20: Advanced Program-to-Program Communication Extensions

Your product offers a number of extensions to assist in establishing communications using client/server processing concepts. These extensions assist in application development between different platforms, but might not be supported by other products which implement APPC.

This section contains the following topics:

APPC Extended Verb Set

The following verbs are available to the APPC in your product:

- &APPC ATTACH_DELAYED
- &APPC ATTACH_IMMEDIATE
- &APPC ATTACH_NOTIFY
- &APPC ATTACH_SESSION
- &APPC CONNECT_DELAYED
- &APPC CONNECT_IMMEDIATE
- &APPC CONNECT_NOTIFY
- &APPC CONNECT_SESSION
- &APPC DEREGISTER
- &APPC REGISTER
- &APPC RPC
- &APPC SEND_AND_CONFIRM
- &APPC SEND_AND_DEALLOCATE
- &APPC SEND_AND_FLUSH
- &APPC SEND_AND_PREPARE_TO_RECEIVE
- &APPC SET_SERVER_MODE
- &APPC START
- &APPC TRANSFER_ACCEPT
- &APPC TRANSFER_CONNECT
- &APPC TRANSFER_REJECT
- &APPC TRANSFER_REQUEST

APPC Transactions

In addition to user-defined transactions, the APPC in your product supports a number of system transactions that assist in NCL communication, especially in the area of client/server processing. These transactions are as follows:

START

This is used to initiate a new NCL process in either the local or a remote system

RPC

This is used to call a procedure in either the local or a remote system in the manner of a remote procedure call

ATTACH

Establishes a standard APPC conversation by attaching an NCL procedure by name, rather than indirectly through a transaction identifier

CONNECT

Establishes a conversation connection to an existing NCL process in either the local or a remote system.

These system transactions are automatically defined in the Transaction Control Table during system initialization, but can be modified by the installation if necessary using the REPTRANS, DELTRANS and DEFTRANS commands.

START Transaction-Remote Process Start

The APPC in your product incorporates the system START transaction that allows any NCL procedure to be started as a new process through the &APPC START request. For example:

```
&APPC START PROC=proc DOMAIN=test ...
```

The system TCT entry for the START transaction is used to complete the conversation setup options, including the transaction security requirements.

Any procedure that is designed to be started through the usual START command can be started in this manner by using APPC. The started procedure need not be aware that it was started through APPC. It has no access to the APPC conversation that established the new process and has no requirement to use any APPC facilities.

The new process can be created in any of the following ways:

- As a dependent process (of the requesting process) in the local region
- As an independent process within the same region as the requesting process
- As an independent process within any APPC region (where authorized) in the local APPC system
- As an independent process within any APPC region (where authorized) in any connected product APPC system

The initiating process can request an immediate indication of the success or failure of the new process creation. It can pass the usual procedure initiation parameters to the target procedure. In addition, any subset of NCL variables, or MDO data, can be copied from the environment of the initiating procedure to that of the new process. Once the request completes all links between the initiating process and the new process are lost, and the new process operates completely independently of the initiating process.

Note: When a process is started as a dependent process, the usual NCL relationships hold true.

Remote Procedure Call (RPC) Transaction

The APPC in your product incorporates the system RPC transaction that allows any NCL procedure to be executed from the context of the calling procedure through the &APPC RPC request. For example:

```
&APPC RPC PROC=proc LINK=link23 PARMS=(A,B,C)
```

The system TCT entry for the RPC transaction is used to complete the conversation setup options, including the transaction security requirements.

Many procedures that are designed to be called from another procedure through the usual EXEC command can operate successfully in this manner using APPC. However, the only context that can be transferred between the calling procedure and the remote procedure, is in the form of NCL variables and MDO data. All other resources in the calling environment (such as files, internal read queues, and so on) remain the strict domain of the calling procedure. The called procedure need not be aware that it was started by using APPC, has no access to the APPC conversation that established the call path, and has no requirement to use any APPC facilities.

The target procedure is started as an independent process in the same or any connected product APPC system, with the context passed. The calling procedure is suspended until the remote procedure completes its execution and terminates. However, the called procedure need not be aware that it was invoked via a remote procedure call.

When the remote procedure terminates, control is returned to the calling procedure with an indication of the success or failure of the request. Shared NCL variable information can be passed back on return. If the called procedure terminates abnormally, or APPC communication is lost to a remote system, the appropriate error information is returned.

The new process can be created in any of the environments allowed for a remote process start.

ATTACH Transaction-Allocate a Procedure

The APPC in your product incorporates the system ATTACH transaction that allows any NCL procedure to be attached and establishes the usual APPC conversation connection. For example:

```
&APPC ATTACH PROC=proc DOMAIN=test
```

This form of allocation proceeds as a special internal transaction. It sets up the conversation with the nominated procedure without the requirement to set up a Transaction Control Table entry for the procedure. The system TCT entry for the ATTACH transaction is used to complete the conversation setup options, including the transaction security requirements.

Note: This form of allocation can be useful between systems where the procedure name is obtained indirectly, but should not be thought of as a general replacement for the use of the Transaction Control Table.

Once started, the conversation is available in the attached procedure, exactly as it would had a standard allocation request been issued. The conversation is operated by both partners in the usual manner.

CONNECT Transaction-Connect to an Active Process

The APPC in your product incorporates the system CONNECT transaction that requests a connection between the a client procedure and any other active NCL process to act as a server. It establishes the usual APPC conversation connection. For example:

```
&APPC CONNECT NCLID=nclid DOMAIN=test
```

This form of allocation proceeds as a special internal transaction, and sets up the conversation between the client and the nominated server process if permitted by the server. The system TCT entry for the CONNECT transaction is used to complete allocation options.

Note: The process must be active or the conversation will terminate with an allocation failure.

Once connected, the conversation is available in the target procedure exactly as it would had a standard allocation request been issued. The conversation is operated by both partners in the usual manner.

APPC Client/Server Processing

APPC is a general application protocol for communication between any two programs in an SNA environment. It can be used to deliver data in substantially one direction-such as a file transfer. It can be used for a complicated dialog-such as a terminal to application datastream. However, it can be easily adapted to implement communication along the client/server processing model.

The usual model of client/server processing is that there are potentially many clients, and usually a smaller number of servers, in a distributed processing environment. The clients request information and they are served that information by a server. A simple question, followed by one or more related responses, is an example of this approach, however the actual dialog could be more complicated.

Ideally the client does not nominate the server, but merely requests a particular type of service, and depending upon configuration options, the transaction will be directed to an appropriate server for processing.

Server Processes

A server is a single NCL process that can accept connections from one or more clients, either serially or concurrently, at the server's discretion. Registration of the server is successful if the server name is unique within some scope, as determined by the server name registration request. Registration of the server name can be tied to process creation such that if the registration is unsuccessful the process creation fails. Alternatively, an executing process can attempt to register itself as a server at any time.

A transaction that starts an NCL procedure can now indicate, through the Transaction Control Table, that, once attached, the target NCL procedure is to be registered as a server process. A server process can also be started by the usual START command before any communication is necessary.

In general, any NCL process, regardless of whether or not it has a registered server process name, can in fact behave as a server process. That is, any active NCL process can accept client connections. Server name registration provides a mechanism for preventing duplication of server processes, but, more importantly, assists in targeting the correct server by supplying a meaningful name. While a server is active, new transactions can target the process and request connection. Such transactions can, at the server's discretion, be queued to the server through the APPC transfer mechanism in your product or automatically connected to the server for immediate operation.

Any process that has a registered server name can be targeted, by defining transactions in the TCT that allocate that particular server name. If the server is not active, the first transaction selecting a TCT entry for that server name will start the server process. Subsequent transactions locate the active server process and queue a connection request. Any user defined transaction can target a server in this manner. In addition, the ATTACH and CONNECT transactions allow a server to be targeted by the requestor.

Client/Server Connection Mode

During server process creation and initialization, all client connection requests that target the process remain pending until the server declares its operational mode for client connection. It can choose to automatically accept new connections, request notification before accepting connections, or reject connections.

Connections only apply to new conversations targeting an active process such as:

- Any standard transaction where the TCT entry indicated a server name as target, and the server is active
- Any ATTACH transaction that targets a server name, and the server is active
- Any CONNECT transaction that always targets an active NCL process

However an explicit conversation transfer request, resulting from an &APPC TRANSFER operation, is not considered a connection request and its mode of operation is unchanged.

The connection mode chosen by the server depends upon whether it intends to serialize connection processing, or operate multiple conversations concurrently. It can also depend on whether the server makes any use of the PIP data sometimes present with new conversations.

Automatic Connection Mode

The server declares it will operate in automatic connection mode by issuing the following statement after process initialization:

```
&APPC SET_SERVER_MODE CONNECT=ACCEPT
```

or

```
&APPC REGISTER SERVER=server CONNECT=ACCEPT
```

Following this statement any pending connection requests, or any subsequent connection requests, are automatically connected to the server and available to operate immediately in receive state. When using this mode no access to the PIP data, if present, for a new conversation is available. Hence this mode of operation is unsuitable for servers that service transactions making use of PIP data.

Since all new conversations connect in receive state, after an automatic connection they would satisfy a receive request that specifies any active clients. For example:

```
&APPC RECEIVE_AND_WAIT ID=CLIENTS VARS=A*
```

or

```
&APPC RECEIVE_NOTIFY ID=CLIENTS
```

Any conversation that connected to the server, that is any client conversation, can satisfy such a receive. However, conversations started by the server, even if they are in receive state, do not satisfy these requests. In either case a new connection, or an existing conversation where more data had just arrived, could satisfy the receive request.

Conversations are serviced in the order that data arrives. The actual conversation satisfying the receive is identified by the usual &ZAPPCxxx system variables. The server, having received some data, can choose to operate that conversation specifically to satisfy the client, before issuing the generic receive against all clients to process the next item of work, thus serializing server activity.

Notification Mode

As an alternative, the server can declare it will operate in notification mode by issuing:

```
&APPC SET_SERVER_MODE ...  CONNECT=NOTIFY
```

or

```
&APPC REGISTER ...  CONNECT=NOTIFY
```

Once this mode is set any pending connection requests, or any subsequent connection requests, are notified to the server by an event message being queued to the process's internal environment. This message is the same as that created by a transfer request from another process, and the server accepts or rejects these connections in the same manner as a transfer request. An &INTREAD statement can be used to receive the notification, informing the process of the conversation identifier that is pending connection. The server can choose to accept the connection request and begin processing the transaction. For example:

```
&APPC TRANSFER_ACCEPT ID=&id VARS=PIP*
```

In this case, it is possible for the server to obtain any PIP data present in the attach request. However, the server might choose to reject the new transaction. For example:

```
&APPC TRANSFER_REJECT ID=&id RETRY=YES
```

This is manifested in the remote allocation system as an allocation failure, with a reason of either:

```
TRANS_PGM_NOT_AVAIL_RETRY
```

or

```
TRANS_PGM_NOT_AVAIL_NO_RETRY
```

If the connection is accepted, the conversation is connected in receive state and is operated in the usual manner. This form is most useful when the server is to continue to operate in notification mode, by using the asynchronous form of receipt. For example:

```
&APPC RECEIVE_NOTIFY ID=CLIENTS
```

Rejection Mode

Servers can optionally mask off further connections by setting the connection mode to reject, as follows:

```
&APPC SET_SERVER_MODE CONNECT=REJECT RETRY=NO
```

In this case the RETRY option can be YES or NO, and subsequent connections are rejected as for a transfer reject. This can be useful where the server is terminating to indicate whether or not processing can be retried.

Transfer a Conversation

Your product supports some implementation-specific &APPC verb options that allow an active conversation to be transferred from one NCL process to another. These are as follows:

```
&APPC TRANSFER_REQUEST
&APPC TRANSFER_ACCEPT
&APPC TRANSFER_REJECT
```

The TRANSFER_REQUEST option requires that a target NCL identifier be specified. This NCL process is then notified of the transfer request by a message queued to its internal environment. It can accept or reject the conversation transfer (as indicated by the syntax shown previously) which completes the transfer request. After the transfer, the requesting procedure has no access to the conversation.

A transfer can only take place while the conversation is in send or receive state. Most likely this is immediately following the completion of an allocate or attach request. Using this technique it is possible to pass additional conversations to a process that is already handling other conversations.

Chapter 21: Using APPC to Communicate with Other Systems

This section contains the following topics:

About APPC

APPC allows programs to communicate and exchange data using a common set of communication protocols. Communication takes place between programs using LU Type 6.2 sessions.

APPC is supported on a wide variety of both IBM and non-IBM platforms. This allows distributed applications to be developed in a heterogeneous networking environment. Although APPC standardizes communication between such applications, the programs themselves can reside on differing hardware and software platforms, and be written in different programming languages.

APPC employs a peer protocol. This means that a program's APPC behavior is not restricted in operation due to the network node where it resides. Nor is the node's communications ability a determining factor in application design.

APPC provides a set of common services available to an application through an architected verb set. The verb set has a direct relationship with underlying LU6.2 session protocols, but the application itself is written in a manner totally independent of SNA sessions. Only those internal services that support the verb set in a particular product implementation are concerned with managing session activity.

Transaction Programs and Conversations

When two programs communicate using APPC, they do so via a communication path called a conversation. All activity that occurs as a result of APPC communication is called a transaction and the programs involved are known as transaction programs. Terms such as client/server processing or distributed transaction processing, are often used to describe this form of data processing.

One feature of APPC is that two programs do not have to be running to start communications. APPC allows a program in any LU to request the invocation of a program in any other LU to fulfill its processing requirement. The program achieves this through a process known as allocation, which establishes a conversation, and sends an attach request that the nominated program be invoked to service the remote LU of the conversation.

The invoked program is said to be attached in the remote LU. An attached program can itself attach other programs. These all form part of the same transaction, but each communication instance between programs is considered a separate conversation.

Within the product region, any NCL procedure can issue an allocation request to start a conversation with another procedure or program. These can be in the same or another network LU. When the region receives an incoming request to attach a program, an NCL procedure is started to service the conversation.

Conversations and Sessions

Each conversation is mapped to an LU6.2 session. While a conversation is active it has exclusive use of that session. An active conversation must terminate before any other can begin on that same session. This means that any one session is seen by LU6.2 conversations as a serially-shared resource.

Concurrent Conversations

To conduct concurrent conversations between any two programs, the use of parallel sessions must be supported between them. The number of concurrent conversations that can be established between programs is dependent upon the number of parallel sessions available.

Note: Applications running under a PU Type 2.0 node, such as PC-based applications, must always play the role of Secondary LU and are restricted to a single LU6.2 conversation at any given time. Use of Node Type 2.1 and independent LUs allows such devices to play the role of a Primary LU (BIND sender) and also allows a parallel session capability.

Share a Single Session

To share a single session for all transactions, conversations should be relatively short-lived. The conversation turnover and available session limit should be considered in program design and session configuration.

Conversation Handling

Within the product region, NCL procedures acting as transaction programs are concerned only with the management of their conversations. All aspects of conversation to session mapping and session management are handled by the LU Services components of your product region and VTAM.

Session Polarity

Each LU6.2 session has a designated polarity indicating which LU is the contention winner. This polarity can be negotiated during the session BIND phase but is then fixed for the session.

The LU designated as contention winner controls conversation allocation activity for the connection. It has the right to initiate conversations at any time without notifying the LU at the other end. Conversely, the LU designated as contention loser must request permission from the contention winner LU before it can start a conversation on that session.

For a parallel session connection the typical configuration is to support enough contention winner sessions in each direction to service concurrent demand for conversation initiation from each LU of the connection.

Conversation Types

There are two types of conversation supported by APPC:

- Basic conversations-for use by SNA Services programs only
- Mapped conversations-support user transaction programs

Both basic and mapped conversations use the LU 6.2 session in the same way. The only difference is the manner in which they are operated at the LU6.2 protocol boundary. At this boundary there is one verb set for mapped conversations, and another for basic conversation. There are also some type-independent verbs.

Your product supports the allocation of conversations as mapped or basic. Because of the high level nature of the NCL verb interface, all conversation operation is equivalent to the LU6.2 mapped conversation verb set.

APPC Option Sets

APPC consists of basic facilities that all implementations must support. In addition to these facilities there are other more advanced facilities that are named option sets. An option set is comprised of a number of optional facilities. Optional facilities are grouped into sets so that the implementation choices are restricted to these sets, and not each individual option.

A full list of all LU6.2 Option Sets is contained in the SNA Transaction Programmer's Reference Manual for LU Type 6.2. The following section describes the option sets that are supported by your product.

APPC Option Set Support

The following option sets are supported:

Conversations between programs located at the same LU

Allows a program to allocate a conversation where the remote program is in the same LU as the local program.

Delayed allocation of a session

Allows a program to allocate a conversation and begin sending data before a session can be assigned to support the conversation.

Immediate allocation of a session

Allows a program to allocate a contention-winner session only if one is immediately available, otherwise the allocation fails.

Session-level LU-LU verification

Provides LU security by designating a password to verify the identity of a remote LU.

User ID verification

Allows a program to allocate a conversation with user verification by means of a supplied password or an already-verified indicator where appropriate.

PIP data

Allows a program to allocate a conversation and supply program initialization parameters to the attached remote program.

Logging of data in a system log

Allows a program to record error information in a system log.

Flush the LU's send buffer

Allows a program to explicitly cause data transmission on a session.

Prepare to receive

Allows a program to change from send to receive state with a number of different options.

Receipt with wait

Allows a program to operate a number of conversations and be notified when information is available on any one of them.

Receive immediate

Allows a program to request any data on a conversation but continue processing if none is available.

Test for request-to-send received

Allows a program to test whether a request to send indication has been received on a conversation.

Data mapping

Allows a program to request the mapping of data by the local and remote LUs.

Get attributes

Allows a program to obtain information about a mapped conversation.

Supported APPC Products

This implementation of APPC lets you communicate with the following products or components:

- Other domains
- CICS/OS/VS V1.7
- OS/2 Comms/Manager
- CommLink SNA/LU6.2
- APPC/MVS

It is possible that other APPC products that have not been tested will also operate with your product.

Define APPC Links

APPC links can be defined in two ways:

- Statically-using the LINK command
- Dynamically-using APPC table definitions

The type of APPC link you define is dependent on your needs. A static APPC link is used for one-off APPC connections. A dynamic APPC link is necessary if you want APPC links to be established on demand.

For more information about static APPC links, see the *Reference Guide*.

Define Dynamic APPC Links

To define dynamic APPC links identify the components that can take part in APPC communication and their operational requirements to the local domain. These definitions are kept in APPC tables maintained by a set of product commands. The table definitions perform the following functions:

- Relieve NCL from the task of fully specifying all the parameters required to initiate a conversation. This simplifies allocation requests.
- Provide validation for both locally and remotely initiated conversation requests. This provides a level of integrity before allowing communication to take place.
- Include information concerning destination attributes relevant to LU6.2 usage and for controlling resources such as sessions. This information allows your product to modify its behavior to suit a variety of LU6.2 implementations.
- Most importantly, these tables allow on-line maintenance of LU6.2 operational facilities, including aspects of control over NCL transactions without resorting to the modification of NCL procedures.

The APPC tables support the separation of the APPC programming facilities from the session support facilities. This allows NCL procedures to be developed and maintained in isolation from the changing network requirements.

There are four control tables that can be defined:

Transaction Control Table (TCT)

Associates transaction program identifiers with transaction program names

Dynamic Link Table (DLT)

Defines those LUs with which the local domain allows LU 6.2 sessions to be established.

Option Set Control Table (OSCT)

Defines those options, supported by a particular LU, that are connected to your product through LU 6.2 session communication

Mode Control Table (MCT)

Defines mode names, the session characteristics they represent, and the number of sessions they support

APPC Table Requirements

Although there are four APPC tables and each supports many parameters, it is not necessary to define all of the tables or parameters. In most cases the system provides default values. The following gives an indication of table definition requirements:

- Transactions for which both ends of the conversation run in the same domain, need only a TCT definition to define for each transaction the name of the NCL procedure that will service the request.
- LU6.2 connections that use the default option set, only need TCT and DLT definitions. This is sufficient to authorize single session connections to service conversations requiring no security and no data mapping support, and is appropriate for typical communication between your product region and work stations running APPC.
- LU6.2 connections that use advanced options (for example, specific logmode) need all four table definitions. Such advanced options are likely to be required to enable sophisticated use of APPC between one domain and other domains or CICS systems.

How APPC Control Tables Interact

An APPC link can be started from either the remote or the local domain as a result of a conversation allocation request (for example, an &APPC ALLOCATE) or an operator command (for example, LINK START). In either case the relevant DLT entry is located using the link name or LU name depending on which one was provided.

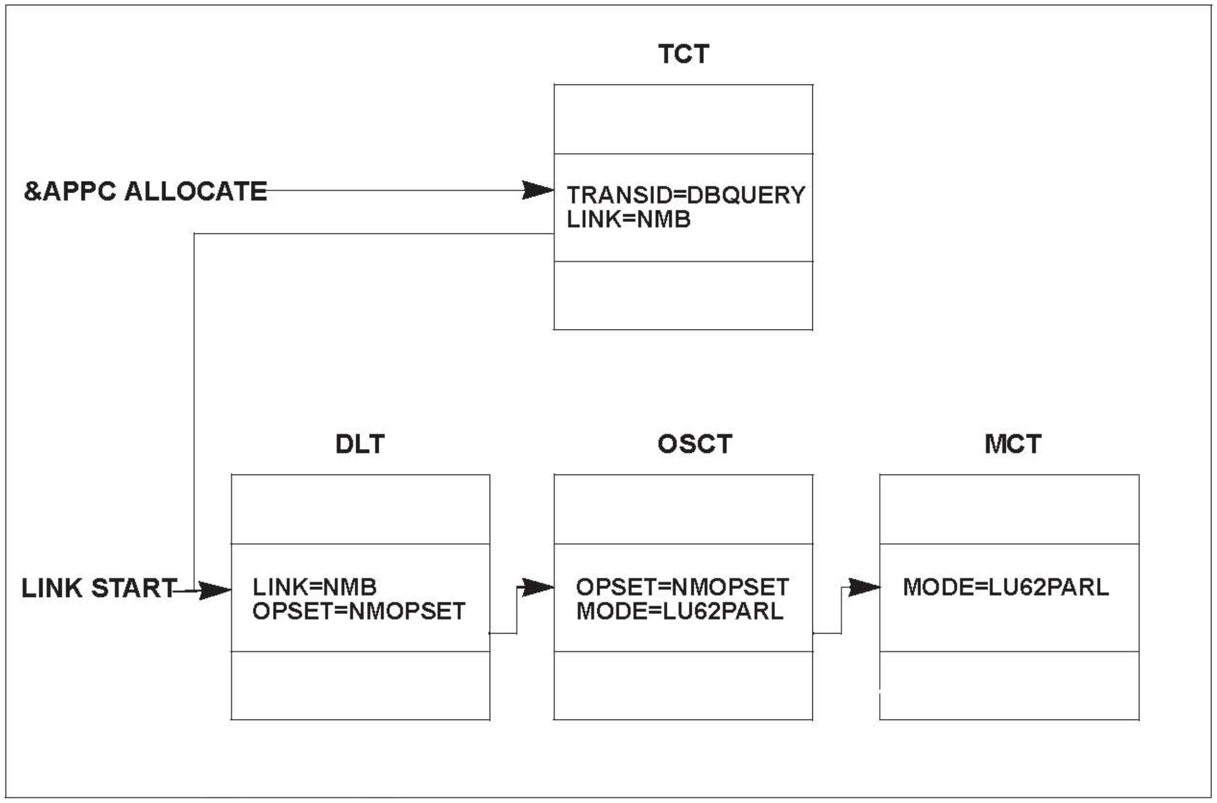
A DLT definition is needed for each remote LU that is to be connected to your product region using LU6.2 sessions.

Link activation processing will then locate the OSCT entry nominated in the DLT definition and its associated MCT entries. The information from these tables is used to activate a link with the desired properties (for example, parallel sessions). If no OSCT or MCT entries are associated with a DLT definition default options and modes are used.

Example: APPC Table Interactions

When an NCL procedure issues an allocation request the transaction ID is used to locate the TCT entry. If no destination information was provided in the &APPC ALLOCATE verb the link name or LU name in the TCT are used (if defined) to determine if the link is active. If the link is not active, the LU name or link name is used to locate the relevant DLT entry and link activation then proceeds as described in the following figure.

This figure show APPC table interactions during the activation of links.



Define APPC Tables

The following sections describe how to define each of the APPC control tables.

Define a Transaction Control Table

The Transaction Control Table (TCT) is used to define all transactions to your product. Each TCT entry can be used to control allocation requests, attach requests, or both. A conversation cannot proceed without referencing a valid TCT entry.

All transaction identifiers and transaction program names (TPN) must be defined in a TCT to relate each transaction identifier to the appropriate TPN. Therefore, when a program issues an allocation request it uses a transaction identifier to indicate which transaction program it wants to communicate with. The system relates this identifier to the TPN.

The TPN is carried in the attach request sent to the remote domain. At the remote domain attach processing will use the TPN to start the correct program to service the request. The TPN is therefore known to both LUs that are to participate in a conversation. Transaction identifiers are only known locally to each LU.

Define Transactions

The DEFTRANS command is used to define a TCT. This command lets you define transactions between your domain and another LU.

Example: Define Transactions

To define a transaction with a transaction ID of DBQUERY and a TPN of DB01, enter the following command:

```
DEFTRANS TRANSID=DBQUERY TPN=DB01
```

When an NCL procedure wants to invoke a transaction with a transaction ID of DBQUERY, the TCT entry is located and the associated TPN is extracted. The TPN and any other information specified in the TCT entry is sent in an attach request to the remote system specified by the NCL procedure on the &APPC ALLOCATE verb.

If you want the transaction to start the NCL procedure DBQRY01 in the remote system, define the name of the procedure by using the PROC operand of the DEFTRANS command. This TCT entry must reside in the remote system.

Define a Qualified Transaction

You can use the same transaction ID to invoke a different TPN name during certain circumstances, such as, testing a new version of an NCL procedure.

Do this by defining a qualified transaction entry in the TCT and specify the same qualifier in the DLT entry that defines the remote LU to the local domain. The qualifier is specified by the QUAL operand of the DEFTRANS and DEFLINK commands.

More information:

Run a Qualified Transaction (see page 415)

Generic Transactions

A transaction identifier can have the same common suffix as the name of the transaction procedure it subsequently invokes-for example, the transaction identifier DBUPDT and the procedure APPCUPDT have the suffix UPDT in common. If there are many transactions starting with a fixed prefix such as DB that invoke procedures with a fixed prefix such as APPC but with the same variable suffix, a single generic transaction definition can be used to service them.

Generic transactions are defined by specifying masks for the transaction IDs, TPNs, and procedures in the TCT entry.

More information:

Run a Generic Transaction (see page 416)

Specify a Default Destination

The destination of an APPC transaction request can be specified by using the &APPC ALLOCATE verb or in the TCT definition. If no destination is specified on the &APPC ALLOCATE verb, the destination in the TCT is used. If no destination is specified on the &APPC ALLOCATE verb or in the TCT, a default destination of ENV=CURRENT is assumed.

If the same destination always services a transaction request, you can specify a default destination in the TCT definition. To specify a default destination in a TCT definition, specify one of the following on the DEFTRANS command:

- A link name using the LINK operand
- An LU name using the LUNAME operand
- A product domain ID, using the DOMAIN operand
- The NCL environment using the ENV operand (for local conversations)

Example: Specify a Default Destination

To define an APPC link with a transaction ID of DBQUERY, a TPN of DB01, and a link name of NMB, enter the following command:

```
DEFTRANS TRANSID=DBQUERY TPN=DB01 LINK=NMB
```

Note: For more information about the options that can be specified in a TCT, see the description of the DEFTRANS command in the online help.

Maintain TCT Definitions

You can replace or delete a TCT definition by using product commands. The REPTRANS command lets you replace an existing TCT definition with new values. The DELTRANS command lets you delete a TCT table definition.

Note: For more information, see the online help.

Define a Dynamic Link Table

A Dynamic Link Table (DLT) is used to support the dynamic addition of different types of links, including INMC and APPC links. In the case of APPC the DLT is used to define those LUs with which the local system will allow automatic link activation. Two types of APPC links can be defined, those that support parallel LU6.2 sessions and those that support only a single LU6.2 session.

To specify a DLT, use the DEFLINK command.

Example: Define a Dynamic Link Table

To allow an APPC link with an LU of NM1, enter the following command:

```
DEFLINK TYPE=APPC LUNAME=NM1 LINK=NM1
```

The DLT supports other options including the specification of a password for link level security.

More information:

APPC Security (see page 403)

Specify an Option Set

To include support for option sets, specify an OSCT definition with the OPSET operand. When an APPC link is activated, the OSCT is then accessed to determine the optional features it is to support.

If an option set is not nominated or the one nominated can not be found during link activation, the system uses a default option set. The default defines a parallel session link with no data mapping support and no conversation level security support for the connected system.

Maintain DLT Definitions

DLTs are maintained by using the REPLINK and DELLINK commands. REPLINK lets you replace a DLT definition. DELLINK lets you delete a DLT definition. For more information, see the Online Help.

Define an Option Set Control Table

An Option Set Control Table (OSCT) is used to define the optional features that the APPC link supports during communication. The option sets provide these optional features.

To define an OSCT, use the DEFOPSET command. You can specify single or parallel session links by using the PARSESS operand. However, if not specified, the default is parallel sessions.

During the activation of a link, the OSCT entry nominated in the DLT definition for the destination is used to determine the supported options and to indicate the associated mode names.

Example: Define an Option Set Control Table

To define an OPSET of NMOPSET with single session links issue the following command:

```
DEFOPSET OPSET=NMOPSET PARSESS=NO
```

Note: For more information, see the description of the DEFOPSET command in the online help.

Nominate Mode Names

An OSCT entry can be associated with up to four mode names. You can use the MODE operand to specify the names of the MCT entries that define the LU6.2 mode names associated with this OSCT entry. Multiple mode names are useful for parallel session links to group sessions with similar characteristics. If no mode names are specified in the OSCT definition, a default mode is used during link activation.

Maintain OSCT Definitions

OSCTs are maintained with the REPOPSET and DELOPSET commands. Use the REPOPSET command to replace an OSCT definition and the DELOPSET command to delete an OSCT definition.

Note: For more information, see the online help.

Define a Mode Control Table

To request a session with certain characteristics specify a mode name on the allocation request or a Mode Control Table (MCT) definition in the OSCT. The MCT definition specified contains an LU6.2 mode name which has associated session characteristics.

To define an MCT entry, use the DEFMODE command.

Example 1: Define a Mode Control Table

To define an MCT entry with the name LU62PARL which references the LU62PARL mode name, enter the following command:

```
DEFMODE MODE=LU62PARL MODENAME=LU62PARL
```

Example 2: Define a Mode Control Table

To define an MCT entry without the MODENAME operand enter the following command.

```
DEFMODE MODE=LU62PARL
```

The name of the MCT entry is used as the LU6.2 mode name.

VTAM Logmodes

To associate an LU6.2 mode name with specific session characteristics (for example, Class of Service, session level pacing), some implementations of APPC require the matching VTAM logmode names with the derived characteristics to be defined to the MCT via the LOGMODE parameter. If omitted, the VTAM default logmode values for those parameters are used.

Note: Your product can successfully initiate sessions using LU6.2 mode names with or without matching VTAM logmode names. However, other implementations, such as CICS and OS/2, require the LU6.2 mode name to match the VTAM logmode name to successfully activate sessions initiated by these products.

Specify Modes for Single Session Links

If you are using single session links, the primary use of the MCT entry is to derive a VTAM logmode name if session establishment is initiated by your product.

Specify Modes for Parallel Session Links

If you are using parallel session links, mode names can be used to partition the available sessions into logical groups, with slightly different characteristics.

Example: Specify Modes for Parallel Session Links

To define a parallel session mode that supports a maximum of ten sessions, a minimum of two contention winner sessions for the local domain, and a minimum of two contention winner sessions for the remote system enter the following command.

```
DEFMODE MODE=LU62PARL SESSLIM=10 MINWIN=2 MINLOSE=2
```

Default Modes

If you do not nominate an option set in the DLT definition, both the default OSCT and the default MCT are used at link activation. The default MCT is the first entry in the MCT or if no entries are defined an internal default applies with MODENAME=APPCMODE, SESSLIM=20, MINWIN=5 and MINLOSE=5, no logmode name is defined. The same applies if you nominate an MCT entry in your OSCT but the nominated modes can not be found.

Maintain MCT Definitions

MCT entries are maintained with the REPMODE and DELMODE commands. The REPMODE command lets you replace an MCT entry. The DELMODE command lets you delete and MCT entry.

Note: For more information about these commands, see the online help.

Chapter 22: APPC Security

APPC links provide the following two levels of security:

- Link level security, or LU-LU verification
- Conversation level security

This section contains the following topics:

Link Level Security

Link level security is a session level security protocol exchanged during session activation to confirm the identity of the partner LU. This confirmation requires the use of an eight-character password known to both LUs that comprise the link. The password is not transferred on the session but used to encrypt data. The other LU analyzes the encrypted data to determine whether the password is correct.

The password is a 2- to 16-byte hexadecimal string. The string is defined on the DEFLINK command or supplied by an operator on a LINK START request. When a password is present, your product invokes LU-LU session verification processing. If verification fails for any reason, then link and session activation fails.

Example: Link Level Security

To specify a password of X'A1B2C3D4' in a DLT definition for SYSB, enter the following command:

```
DEFLINK LINK=B LU=SYSB TYPE=APPC PASSWORD=A1B2C3D4
```

If PASSWORD is not specified when defining the DLT and a password is not given in the LINK START command, LU-LU verification is not performed.

Conversation Level Security

Conversation level security is used to verify the identity of the partner transaction program. The relevant security information (for example, user ID and password) is sent in the attach request and is verified by the receiving LU. If the correct security information is not supplied the request is rejected.

A TCT is used to define the level of conversation security required by a given transaction. When an NCL procedure issues an allocation request the TCT entry is used to determine the level of security information that needs to be included in the attach request to be sent to the remote LU.

Run a Secured Transaction

To run a secure transaction define the SECURITY operand of the DEFTRANS command in a TCT entry. Options on the operand are: NONE, SAME, USER, and USERPSWD. The default is NONE which means that the transaction runs without security.

Example: Run a Secured Transaction

To secure the transaction for DBQRY01 with the user's password, enter the following command:

```
DEFTRANS TRANSID=DBQRY01 TPN=DB01 SECURITY=USERPSWD
```

When an allocation for the DBQRY01 transaction is issued, the password for the user environment is passed to the destination for verification.

Security Options

If you specify SECURITY=SAME in the TCT, the allocation assumes the security of the request source. If you specified the user ID and password on the allocation request, it assumes the same as if you had specified SECURITY=USER. If you do not specify any user information, it assumes the same as if you had specified SECURITY=NONE.

For allocation requests, security is performed for a transaction under the following conditions:

- The NCL procedure which issued the request is running in a signed-on environment
- The USERID and PASSWORD operands were specified in the allocation request

Otherwise it will run as an unsecured transaction.

Incoming attach requests that contain security information result in a secured transaction and if no security information is present the transaction runs as unsecured.

Conversation Level Security and NCL Procedure Environments

NCL procedures running in a signed-on user environment can issue allocation requests for secured transactions without specifying their user ID and password. If an NCL procedure is running in an environment that is not protected by a user ID and password signon, such as EASINET, it can only request allocation of secured transactions by specifying the USERID and PASSWORD operands in the allocation request.

For incoming attach requests the TCT security level determines the type of NCL environment in which the NCL procedure nominated to service the transaction is run. An attach request for an unsecured transaction results in the NCL procedure being started in the background server (BSVR) region.

If the request is for a secured transaction, a special APPC region is created (if none already exists) for the user. The user is signed on using normal security procedures. This happens before the transaction can be started.

Secured transactions require a signed-on environment before execution is permitted. An attempt to start such a transaction in some other environment fails with a security error return code.

Specify Security for APPC Links with Remote Systems

The OSCT is used to indicate the security options that can be accepted from a remote LU. To specify security parameters, use the SECURITY operand of the DEFOPSET command. The options for the operand are: NONE, USER, and USERPSWD.

If you specify NONE, the local system does not process any security information sent by the remote LU in an attach request. The remote LU can invoke unsecured transactions only.

If you specify USERPSWD, user IDs and passwords are accepted from the remote LU and passed to the local security system for verification. However, the already verified indicator is not accepted.

If you specify USER, then USERPSWD is assumed and in addition the already verified indicator is also supported. If the indicator is set, a request for user ID authorization is made to the local system to validate the region without a password.

APPC Region Use

NCL procedures, defined in the TCT to service secured transactions, execute in a special APPC region built-in response to an attach request for a specific user ID. Normally all attached requests for the same user ID execute in the same APPC region. When the last NCL procedure completes execution the region is deleted.

To restrict the building of an APPC user region to a given user ID, specify a DLT entry using the USERENV operand of the DEFLINK command. The options for the operand are GLOBAL or LINK.

If GLOBAL (the default) is specified, an APPC region is built to service all attach requests from the same user ID from any remote LU.

If LINK is specified, an APPC region is built to service attach requests, for a given user ID, received from the remote LU defined by the DEFLINK definition.

Activate APPC Links

You can activate a link in the following ways:

- Manually-via a LINK START command
- Automatically-as the result of an allocation request

Start Links Manually

To start an APPC link manually use the LINK START command.

Example: Start Links Manually

To start an APPC link from SYSA, issue the following command:

```
LINK START=NMB TYPE=APPC
```

Deactivate Links Manually

Regardless of the way an APPC link is started it remains active while any sessions are active even when all conversation activity ceases. If all sessions on the link have closed then the link will automatically deactivate.

To stop the link at any other time, use the LINK STOP command. You can specify different options (DRAIN, QUIESCE, and FORCE) to influence the way the link deactivates.

If DRAIN is used (the default), all conversation activity is serviced before closing the link. If QUIESCE is used, active conversations are allowed to complete normally, but pending conversations complete with an allocation error. If FORCE is used, all conversations are terminated (with a resource failure return code) and link deactivation proceeds immediately.

Example: Deactivate Links Manually

To deactivate the link to NMB by using the LINK STOP command with the DRAIN option, enter the following command:

```
LINK STOP=NMB TYPE=APPC
```

Start Links Automatically

A request to establish an APPC link may come from:

- The remote LU
- The local domain via a conversation request
- An operator request

Provided the correct definitions are in place any program issuing an allocation request can cause link activation if the link is not already active.

The LU name of a network node must be defined in the DLT (see page 399) to authorize automatic LU6.2 session activation between the remote LU and the local domain.

Session Initiation Requests

For a link to be activated automatically, a request to initiate an LU6.2 session must come from a source in the network. All requests can be categorized as coming from one of three sources:

- An external logon request
- An external BIND request
- An internal NCL procedure request

Whenever the first session between your product region and some other network node is requested, the Dynamic Link Table is examined to determine whether the link is authorized, and establish any special options for the link.

External Logon Request

An external logon request arises when these three conditions occur:

- An LU in the network requests a session between the product region and either itself, or some other LU
- The product region is to be the primary session partner
- The logmode specified contains LU6.2 session parameters

Successful processing of such a request results in a BIND being issued to the target LU. The target LU can accept the BIND as it is, negotiate certain BIND parameters, or reject the BIND outright. If not rejected, the negotiated (or unchanged) BIND response is returned to your product region and, if acceptable, the session establishment is complete.

External BIND Request

An external BIND request arises when these three conditions occur:

- An LU in the network requests a session between the product region and itself
- The product region is to be the secondary session partner
- The BIND contains LU6.2 session parameters

Successful processing of such a request results in your product accepting (or negotiating) the BIND from the LU, and the session is established.

Internal NCL Procedure Request

The internal NCL request, &APPC ALLOCATE, can be sourced from any executing NCL procedure to request an APPC conversation with some remote LU. If no suitable sessions are available to that LU, and the session limit (as defined in the MCT) has not yet been reached, then the product region issues a session setup request to VTAM using the logmode name from the MCT, whereupon session initiation proceeds as for an external logon request.

Note: Wherever your product initiates a session, it is the primary session partner, and no logmode is required because the LU6.2 BIND is formatted according to the options for the link.

Sessions where your product region is the secondary session partner must be initiated by the other (primary) session partner. This does not restrict any session usage as it is a peer relationship, and aspects regarding session polarity (such as which LU is contention winner) are negotiable.

How Session Establishment Works

Regardless of the source of the session initiation request, and whether your product region is to be the primary or secondary session partner, session establishment proceeds through a number of phases:

- Phase 1. Session parameter validation ensures that the BIND to be sent (when processing a logon request), or BIND received (when processing an unsolicited BIND), is a valid LU6.2 BIND within the parameters supported by the product region.

  If necessary, the BIND parameters are modified before being sent to the secondary LU, or negotiated with the primary LU, so that they are within the bounds supported by the product region.

- Phase 2. The unqualified network LU name provided by VTAM is extracted. If other LU6.2 sessions already exist with this network node, or are in the process of being established, then Step 3 is performed; otherwise Step 4 is performed next.

- Phase 3. Where other LU6.2 sessions exist, or are activating, to the destination involved, additional checks are made to determine whether the new session is acceptable within the session limits determined by the MCT entries (see Step 5). If so, the session is accepted as a parallel session without further processing; otherwise it is rejected.

- Phase 4. Where no other LU6.2 sessions exist, and none are being activated, to the destination involved, the unqualified network LU name is used to locate a matching entry in the DLT. If an entry cannot be located the session initiation is rejected. Otherwise the DLT details are extracted and used to initialize the APPC link connection. If an OSCT entry was nominated in the DLT, it is located and the processing options are extracted.
- Phase 5. The OSCT entry determines whether this link supports parallel sessions or not. Any mode names listed for the OSCT entry are used to locate the corresponding MCT entries, and the details of each MCT is extracted to form a valid MODE name list for this destination. For each mode name, this list contains the session limits for that mode, as well as other session control information.

Note: If a session initiation request comes from an &APPC ALLOCATE verb, Phase 4 and 5 are performed first, before the session setup request is passed to VTAM. Following this, and for all other session initiation types, processing begins at Phase 1 when VTAM notifies the region of the arrival of a logon request or BIND request.

Single Session Links

When the OSCT entry indicates that parallel sessions are not supported, an external BIND request to establish a parallel session link is rejected. Only a single session link can be established if the OSCT entry has been defined with PARSESS=NO. Hence only one conversation at a time can be active on such a link, and other conversation requests must queue, pending session availability.

If the session setup request comes from the local domain (via a LINK command or NCL procedure allocate request), the product region attempts to set up a contention winner session. However, the polarity of the session can be negotiated by the secondary LU. If the session setup request comes from outside the local domain, the region accepts the session polarity without negotiation.

Parallel Session Links

When the OSCT entry indicates that parallel sessions are supported, an external BIND request to establish a single session link is accepted, and a single session link is set up. An external BIND request to establish a parallel session link is accepted and session limits negotiation is initialized.

For a parallel session link, the product region initializes the link session limit as the sum of the session limits for each mode defined for the link. An attempt is made to initially activate sessions within each mode to at least reach the minimum contention winner counts for the mode. Additional sessions may be activated if conversation demand cannot be immediately serviced by the available sessions.

During link initialization a special transaction known as Change Number of Sessions (CNOS) is executed for parallel links. This transaction negotiates the session limits for the defined modes before sessions are established. The CNOS transaction may be subsequently invoked on demand, such as by remote LU request, to renegotiate session limits at any stage of link operation.

Session Selection for Conversations

When an allocation request targets a particular link an attempt is made to locate a session to be used by the conversation. If an idle contention winner session is located then it is assigned. If not, the conversation goes into a pending allocation state, awaiting a session for the conversation.

For a pending allocation on a parallel session link, further efforts are made to isolate a usable session. Where the number of contention winners can be increased (according to the session limits) then a new session is started. Otherwise, if an idle contention loser session is located, a bid is made to use that session.

For a pending allocation on a single session link, if the session is a contention loser session and it is idle, a bid is made to use that session.

In general, when a conversation enters a pending allocation state, one of three events can occur to provide a session for its use:

- An active conversation ends, freeing up a contention winner session for the pending conversation.
- A new contention winner session starts and becomes available for the pending conversation.
- A response to previous bid indicates that the remote end will allow the conversation to commence on a contention loser session.

Conversations remain pending until one of these events frees a session for use, or until the allocation request is otherwise canceled. An allocation may time out (if a time-out period is specified on the allocation), or a link termination condition may cause the cancellation of all pending requests for the link.

Deactivate Links Automatically

APPC links deactivate automatically when all sessions on the link have been closed.

APPC Link Definition Examples

The following examples illustrate typical APPC table definition requirements to establish LU6.2 connections between the following:

- Two domains (SYSA and SYSB)
- Two domains (SYSA and SYSB) that require security verification
- The product region and a remote LU
- Two domains (SYSA and SYSB) using a qualified transaction
- Two domains (SYSA and SYSB) using a generic transaction

Each example uses a database query application with transaction identifier DBQUERY, a TPN of DB01, and a server NCL procedure DBQRY01.

When defining mode names in the examples, the MODENAME parameter has been omitted in the DEFMODE command, so that the LU6.2 mode name defined is the same as the name of the MCT entry (MODE parameter).

Run Conversations Within the Same Domain

The two LUs of an APPC conversation are run in the same domain to develop a database query application and test it. No sessions are established, no session related definitions are needed, and session overheads are eliminated. Only the TCT entry needs to be defined, with the name of the local domain as the destination LU.

To define an APPC link within the same domain, SYSA

1. Define the following TCT entry in SYSA:

   ```
   DEFTRANS TRANSID=DBQUERY TPN=DB01 LU=SYSA PROC=DBQRY01
   ```

2. Issue an allocation request from an NCL procedure in SYSA for DBQUERY.

The allocation processing locates the TCT entry and determines that the destination is the local domain. This has the effect of starting DBQRY01 in SYSA. During this processing there is no need to reference any of the session related tables DLT, OSCT or MCT.

Run with Already Verified Security

An APPC link is to be established between two domains SYSA and SYSB. The link is to service transactions that may require data mapping support.

To define a link between SYSA and SYSB with data mapping support

1. Define the following DLT entries in SYSA and SYSB that specifies an OSCT entry that supports data mapping:

   ```
   DEFLINK LINK=L01 LU=SYSB TYPE=APPC OPSET=NMMAP (in SYSA)
   ```

   ```
   DEFLINK LINK=L02 LU=SYSA TYPE=APPC OPSET=NMMAP (in SYSB)
   ```

2. Define the following OSCT entry, NMMAP, in both domains:

   ```
   DEFOPSET OPSET=NMMAP MODE=MODELU62 SEC=USER MAP=YES +
   ```

   ```
   LOG=YES PARSESS=YES
   ```

   SEC=USER indicates that the already verified indicator is to be accepted from the remote domain

   Note: If you want the link to be a single session link, specify the PARSESS=NO operand and a MODE that supports single session links in the OSCT entry.

3. Define the following MCT entry, MODELU62 in both domains:

   ```
   DEFMODE MODE=MODELU62 SESSLIM=4 MINWIN=1 MINLOSE=0
   ```

   The LU6.2 mode name defined is MODELU62 which supports a maximum of 4 sessions.

4. Define the following TCT definitions in each domain to allow the invocation of the DBQURY01 NCL procedure:

   ```
   DEFTRANS TRANSID=DBQUERY TPN=DB01 LINK=L01
   ```

   ```
   MODENAME=MODELU62 (in SYSA)
   ```

   ```
   DEFTRANS TRANSID=DBQUERY TPN=DB01 PROC=DBQRY01 (in SYSB)
   ```

   A default mode name for conversation allocation has been specified.

5. An NCL procedure in SYSA issues an allocation request for DBQUERY.

   The TCT entry is located and an attach request for DB01 built. The link name L01 in the TCT is used to locate the DLT and to determine the LU name of the destination, SYSB. The OSCT entry NMMAP nominated in the DLT and the corresponding MCT entry MODELU62 nominated in the OSCT are also located. If the link is not already active the DLT entry is used to start link activation and a parallel session link with data mapping support is established.

   Once the link is active and a session is available the attach request for DB01 is sent to SYSB. Attach processing finds the TCT entry for DB01 and the NCL procedure, DBQRY01 is started.

Run an APPC Link Between a Domain and a Remote LU

An APPC link is to be established between a domain and an OS/2 work station. The link is to service transactions with no data mapping requirements.

OS/2 requires that the mode name match a valid LU6.2 VTAM logmode, therefore in addition to a DLT, you need to define an OSCT and its associated MCT.

To run an APPC link between your domain and an OS/2 workstation

1. Define the following DLT in your domain:

   ```
   DEFLINK LINK=WRKS01 LU=OS2001 TYPE=APPC OPSET=WRKSOPST
   ```

   WRKSOPST is the name of the OSCT entry to be used.

   Note: Ensure that you enable APPC links in your OS/2 system.

2. Define the following OSCT entry in your domain:

   ```
   DEFOPSET OPSET=WRKSOPST MODE=LU62SESS LOG=YES MAP=NO
   ```

   LU62SESS is the name of the MCT entry to be used.

   Note: If you want the link to be a single session link, specify the PARSESS=NO operand and a MODE that supports single session links in the OSCT entry.

3. Define the following MCT entry in your domain:

   ```
   DEFMODE MODE=LU62SESS LOGMODE=LU62SESS SESSLIM=20 +

   MINWIN=1 MINLOSE=0
   ```

   The LU6.2 mode name defined is LU62SESS and supports a maximum of 20 sessions. A logmode has been specified and is the same as the LU6.2 mode name LU62SESS. OS/2 requires mode names to match a valid VTAM logmode entry.

4. Define the following TCT entries so that the DBQUERY transaction can be invoked from the workstation and the database interrogated in SYSA:

   ```
   DEFTRANS TRANSID=DBQUERY TPN=DB01 PROC=DBQRY01
   ```

5. A program running in the workstation issues an allocation request for TPN DB01.

Your domain receives an LU6.2 session request from LU OS2001. The LU name is used to locate the DLT for LU OS2001. The associated OSCT and MCT are also located. A parallel session link with no data mapping support is established. The workstation sends an attach request for DB01 to your domain. The TCT for DB01 is located by your domain and the procedure DBQRY01 is started to service the transaction.

Run a Qualified Transaction

A database query application runs between two domains SYSA and SYSB. The DBQUERY transaction is invoked from SYSA to send an attach request for DB01 to SYSB, where the NCL procedure DBQRY01 is run. A new version of this procedure (DBQRY01T) needs to be tested using TPN DB01TEST and the transaction qualifier TEST.

To use the APPC link to test the new NCL procedure

1. Define the qualifier TEST in the DLT entry in SYSA by issuing the following command:

   ```
   DEFLINK LINK=B LU=SYSB TYPE=APPC QUAL=TEST
   ```

   Ensure that you stop and restart the link to put this qualifier into effect.

2. Define the qualified entry in the TCT entry in SYSA by issuing the following command:

   ```
   DEFTRANS TRANSID=DBQUERY TPN=DB01TEST QUAL=TEST
   ```

3. Define a TCT entry in the remote system (SYSB) for the procedure being tested by issuing the following command:

   ```
   DEFTRANS TRANSID=DBQUERY TPN=DB01TEST PROC=DBQRY01T
   ```

4. Issue an allocation request for DBQUERY.

The existing unqualified TCT entry is used to resolve the name of the destination. The destination name is used to determine that the qualifier TEST is in effect (as defined in the DLT). The TCT entry for DBQUERY qualified by TEST is located and used to complete the allocation request. An attach request for DB01TEST is sent to SYSB where the TCT entry for DB01TEST is found and DBQRY01T is run.

Run a Generic Transaction

Two domains SYSA and SYSB are connected via an APPC link. Transactions starting with the prefix DB (for example, DBQUERY, DBPUT, DBUPDT) are to be allocated in SYSA. The NCL procedures to service these transactions start with the prefix APPC (for example, APPCQRY, APPCPUT, APPCUPDT). This can be done using a single generic TCT entry.

To define a single TCT entry to service all of the above transactions

1. Define the following TCT entry in SYSA:

   ```
   DEFTRANS TRANSID=DB* TPN=TP* LINK=B
   ```

2. Define the following TCT entry in SYSB:

   ```
   DEFTRANS TRANSID=DB* TPN=TP* PROC=APPC*
   ```

3. Issue an allocation request in SYSA for transaction ID DBUPDT.

This matches the TCT entry DB*. The matching characters in the transaction name are removed and the remaining characters (UPDT) are appended to the generic TPN (TP*) to get TPUPDT. This is the TPN sent to the remote domain. It matches the TCT entry for TP* and again the matching characters are removed. The remaining characters UPDT are appended to the generic procedure name APPC* to form the NCL procedure name APPCUPDT to be started.

Chapter 23: Program-to-Program Interface

This section contains the following topics:

Uses of PPI

Program-to-Program Interface (PPI) provides a general-purpose facility for programs, written in any language, to exchange data. It also provides a facility for any program to forward a generic alert to your product.

As PPI is available to any environment, not just NCL, PPI provides a simple, powerful technique to access your product from outside.

For example, an NCL process could provide a batch program with the ability to issue selected product commands and return the results of the command to it.

No special authorization is required to use PPI, and it does not depend on having your product running. The PPI implementation can use either Cross-Memory Services or Service Request Block (SRB) scheduling.

The NCL &PPI verb provides access to the Program-to-Program Interface in your product. This interface allows any programs, executing on the same system, written in almost any programming language to freely exchange information.

The API provided by the Program-to-Program Interface is described in IBM's Tivoli NetView Application Programming Guide: Program to Program Interface.

The PPI services can be provided by the subsystem interface for your product (SSI) or by the Tivoli NetView Subsystem Interface.

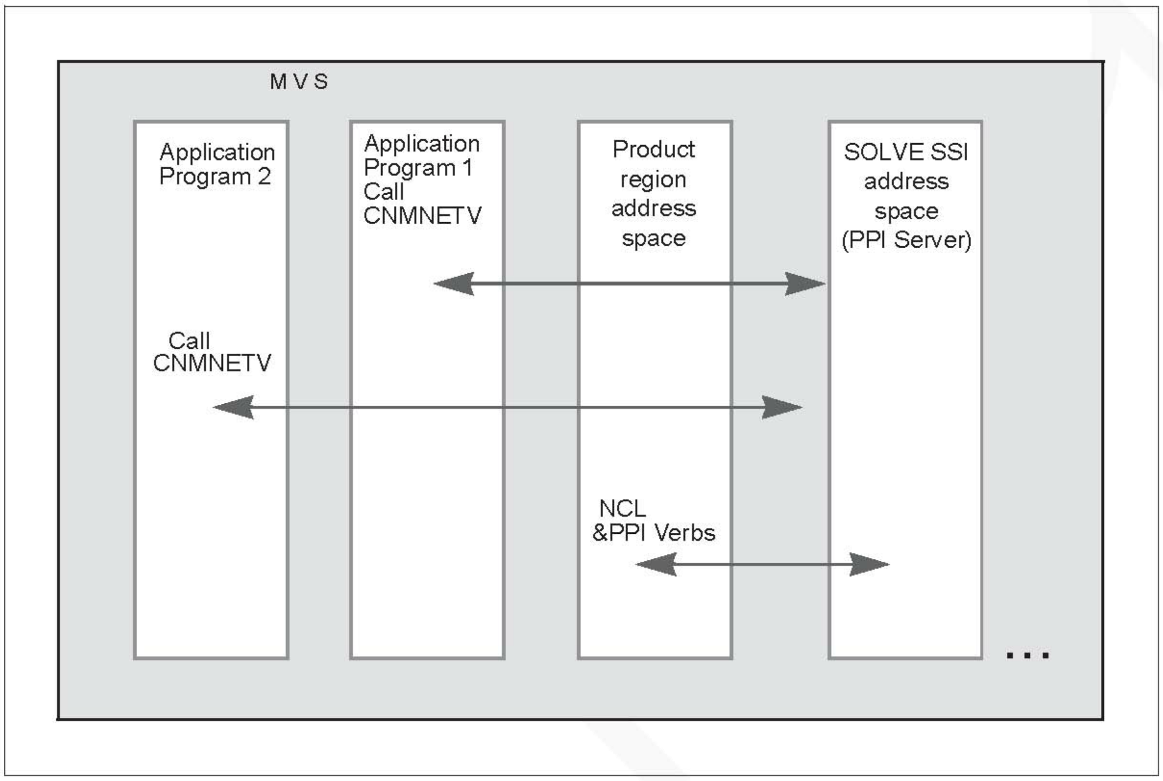
CNMNETM Module

The IBM Tivoli NetView Application Programming Guide: Program to Program Interface discusses the CNMNETV module that you call when using the PPI API. A similar module, CNMNETM, with an alias of CNMNETV, is provided. This module can be loaded by application programs, or link-edited with them. It is fully re-entrant and can be placed in the PLPA, if so desired.

At this time, the IBM version and this version of the module are incompatible with each other's implementation of the PPI. That is, the IBM CNMNETV module will not work in your product's PPI environment, and conversely.

A subsystem is available for your product, which runs as a separate job or started task. This allows PPI to stay active regardless of whether your product is active or not. A command control interface between the subsystem and your product allows control of the subsystem from any suitably authorized product command environment.

Structure and Data Flow


M V S
Application Program 2
Call CNMNETV
Application Program 1 Call CNMNETV
Product region address space
NCL &PPI Verbs
SOLVE SSI address space (PPI Server)
. . .

As this illustration shows, programs issue calls to CNMNETM (CNMNETV) to send data to the nominated receiver program. The data is buffered in the SSI address space in queues associated with each receiver. When data is available for a given receiver, your product posts an ECB that the receiver can wait on. The receiver can then call CNMNETM using a receiver function to obtain the next data buffer.

Note: Data is not directly moved from sender to receiver. The buffering allows the sender or receiver to run asynchronously.

Interface Details

A brief overview of the interface follows.

- Programs construct a request parameter block (RPB) that contains details about a call to PPI.
- A program then issues a CALL to CNMNETV (CNMNETM), passing the RPB as the only parameter (standard linkage, R1 pointing to the word containing the RPB address).
- Upon return, the RC field in the RPB contains the return code. Depending on the call and the return code, other data can be provided.

  For more information about the correlation between the &ZFDBK and &RETCODE system variables, see the &PPI verb description in the *Network Control Language Reference Guide*.

The following functions can be provided in the RPB function code field:

Function code 1-inquire about PPI status

This function code allows an application program to determine if PPI is available and active in the system. A return code of 10 indicates that it is available.

Function code 2-obtain receiver status

This function code allows an application program to determine the status of a named receiver program. Return codes indicate whether the receiver is defined or not, and whether it is presently active or not.

Function code 3-obtain ASCB and TCB addresses

This function code allows programs written in languages that do not support the ability to obtain ASCB or TCB addresses, the addresses of these control blocks. They are needed for other PPI calls.

Function code 4-define and initialize receiver

This function code allows a program to define itself as a receiver. The program provides a unique 1 to 8 character receiver name and queue limit. Following this call, the receiver is defined, and other programs can send data to it.

The receiver can optionally be defined as authorized. This prevents programs that are not APF authorized from sending data to it.

Function code 9-deactivate a receiver

This function code allows a defined receiver program to deactivate itself. The queue limit can be optionally altered. If a receiver program (Task) terminates without issuing this call, it is automatically deactivated.

Function code 12-send a generic alert

This function code allows any program to send a generic alert to your product. It is sent to the defined receiver NETVALET, which receives these generic alerts in NMVT format, and forwards them on to NEWS.

Function code 14-send a data buffer

This function code allows any program to send a data buffer to any defined receiver (active or inactive). The receiver will be notified if necessary.

Function code 22-receive a data buffer

The function code allows a receiver program to receive the next available data buffer. If no buffers are available, a return code informs the receiver.

Function code 24-wait on an ECB

This function code allows a program written in a language that does not support ECB waiting, to wait for data to arrive.

Function code 60-obtain a unique name

This function code, available only in this implementation of PPI, allows a program to obtain a unique 8 character ID. Although programs that only send data need not have a unique ID, all receivers must have a unique ID. This service is provided to allow a sender to obtain a unique ID so that it can register itself as a receiver for a two-way conversation.

&PPI Verb

All PPI facilities can be accessed using the &PPI verb. A keyword immediately following the &PPI verb identifies the specific request. These keywords generally correspond to the various functions described in the API.

The full set of &PPI requests is:

- &PPI ALERT
- &PPI DEACTIVATE
- &PPI DEFINE
- &PPI RECEIVE
- &PPI SEND
- &PPI STATUS

Note: For more information about the &PPI verb, see the *Network Control Language Reference Guide*.

Return Codes, System Variables, and User Variables

Following each execution of the &PPI verb, the &RETCODE and &ZFDBK system variables are set to reflect the success or otherwise of the request. &RETCODE contains a normalized return code. &ZFDBK contains the PPI return code.

Note: For more information about the correlation between the &ZFDBK and &RETCODE system variables, see the &PPI verb description in the *Network Control Language Reference Guide*. A table shows the returned values of each of these system variables.

Other system variables are:

&ZPPI

Indicates whether this system appears to support PPI or not.

&ZPPINAME

Contains the PPI receiver ID that this NCL process is registered as.

Some &PPI functions set specific NCL user variables:

&PPISENDERID

Set to the PPI ID of the sender of a received message.

&PPIDATALEN

Set to the actual received data length, in bytes.

Determine PPI or Receiver Status

The STATUS option of the &PPI verb allows an NCL process to determine the status of PPI itself (available or not), or the status of a PPI RECEIVER (by using the ID=name operand).

In either case, the process can examine the &RETCODE and &ZFDBK system variables after the request. If &RETCODE is 0, then PPI or the receiver is available or defined.

Define the Process as a Registered PPI Receiver

By using the DEFINE option of the &PPI verb, an NCL process can register itself as a receiver. A 1- to 8-character name can be supplied, which must be unique (that is, not presently defined to PPI or currently inactive). If your product is providing PPI services, an alternative is to use the ID=* option, which causes PPI to provide a unique name. This option is useful when talking to globally named servers, as you need not worry about trying to find a unique name.

A process need not be defined to send data using the SEND and ALERT options. In this case, a sender ID of #nclid (7 characters, for example #001352) is used.

Send a Generic Alert

One function of the PPI facility is to collect generic alerts and forward them to general CNM reporting (for example, NEWS). The &PPI ALERT verb allows any NCL process to send an alert to CNM. The alert must be formatted as an NMVT, including the NMVT header.

Send Data to a Receiver

The &PPI SEND verb option allows any NCL process to send data to a nominated receiver. This receiver must be defined, but can be inactive (in which case data is queued unless the queue limit is reached).

Note: The receiver cannot be an NCL process at all and can reside in another address space.

The data to be sent can be a character string, HEX data that is packed before sending, or an MDO.

Receive Data

An NCL process can receive data directed to its defined receiver ID using the RECEIVE option of the &PPI verb. That data can come from other NCL processes, including other systems, or from other programs.

Standard parsing options, as on the other &xxxREAD verbs, can be used. Alternatively, MDOs can be received.

The WAIT operand allows the procedure to indicate whether or not it will wait if no data is available, and, if no data is available, how long it will wait. Alternatively, the process can use WAIT=NOTIFY, to cause a message to be delivered to the dependent response queue when data arrives, thus allowing other work to be performed. When the notification arrives via &INTREAD, the process can reissue the &PPI RECEIVE.

Deactivate the Receiver ID

The &PPI DEACTIVATE option allows an NCL process to disconnect itself from a defined PPI receiver ID. Optionally, a queue limit can be specified, allowing data to be queued even though no receiver is present. The ID can be reactivated later, by this or any other NCL process.

If an NCL process that is defined to PPI terminates, an automatic deactivation occurs.

Access PPI Facilities

To access PPI facilities, use the &PPI verb. A keyword specified after the verb identifies each facility. The facilities take the form of requests that can be sent to other programs, including:

- &PPI ALERT
- &PPI DEACTIVATE
- &PPI DEFINE
- &PPI DELETE
- &PPI RECEIVE
- &PPI SEND
- &PPI STATUS

Note: For more information about the &PPI verb and its use, see the *Network Control Language Reference Guide.*

Access PPI from NCL Processes

To access PPI facilities in NCL, use the &PPI NCL verb and two NCL system variables, &ZPPI and &ZPPINAME.

Note: For more information about these verbs and system variables, see the *Network Control Language Reference Guide*.

After an &PPI verb is executed from an NCL process, the &RETCODE NCL system variable contains the return code that indicates success or failure. This value can be used as a quick test to check whether the operation worked or not. Generally, 0 indicates that there are no problems, 4 is a warning, 8 or 12 indicate errors of some sort. Return code 20 is only returned after &PPI RECEIVE WAIT=NOTIFY if a notification message arrives later.

If an error code is returned in &RETCODE, the &ZFDBK system variable can be used to investigate the error further. &ZFDBK contains the actual PPI return code as returned in the RPB after a call, or a simulated return code if NCL detected an error condition itself.

Any NCL process in the system can use PPI services with the following restrictions:

- To receive data, a process must register itself to PPI with a unique name, using the &PPI DEFINE option.
- If a process does not register itself, and sends data, the PPI sender ID used is the 6-digit NCLID (leading zeros) prefixed with a hash (#) character.
- An NCL process can be defined as only one receiver at a time.

NCL PPI System Variables

The following system variables are available with the NCL PPI facility:

&ZPPI

Indicates whether or not PPI is available in this system.

&ZPPINAME

Contains the defined receiver ID of the current NCL process.

&RETCODE

Contains a normalized return code after &PPI verb execution. For more information, see the &ZFDBK system variable.

As &RETCODE is set by many NCL statements, it should be inspected immediately after an &PPI statement, or saved in a user variable for later inspection.

&ZFDBK

Contains the actual PPI RPB RETCODE value after an &PPI verb execution. It can be inspected as required when &RETCODE indicates an error condition to determine the exact error.

As &ZFDBK is set by many NCL statements, it should be inspected immediately after an &PPI statement, or saved in a user variable for later inspection.

&ZVARCNT

Set after a successful &PPI RECEIVE to indicate the number of NCL tokens that have had data placed in them.

NCL PPI User Variables

The following NCL user variables can be set by some &PPI verb operations:

&PPISENDERID

Set to the sender ID of a data buffer after a successful &PPI RECEIVE operation. The sender ID can contain ampersand (&) characters.

&PPIDATALEN

Set to the actual byte length of the received data buffer after a successful &PPI RECEIVE operation.

The &PPI RECEIVE also sets nominated user variables with the received data.

Access PPI Facilities from Other Programs

To access PPI facilities from programs other than this product, you construct calls to the PPI. To make a call, you have to know the code for the request you are making, construct a Request Parameter Block (RPB), and specify the PPI API.

IBM's PPI uses an API module named CNMNETV to manage the data that is exchanged between programs. This product provides CNMNETM as the PPI API which has an alias of CNMNETV.

Make PPI Calls

To make a PPI call

1. Construct a request parameter block (RPB) in a block of storage.
2. Make the call.

Construct an RPB

All PPI calls require an RPB. An RPB is a block of storage and must be exactly 56 bytes long. An RPB describes the request you want to make of the PPI. The fields specified in the RPB depend on the type of request you are making. PPI uses other fields in the RPB to return data to your program.

The following table shows the request parameter block (RPB) format and describes the fields that can be used to construct the RPB:

Note: Some fields overlap.

Bytes	Name	Description
00–03	RPBLEN	A binary fullword that must contain the length of the RPB. This field must be set to 56 (decimal or 38 hexadecimal). For compatibility with earlier releases of PPI, a length of 40 (28 hexadecimal), 46 (2E hexadecimal), or 52 (34 hexadecimal) is permitted.
04–05	REQUEST	A binary halfword that must be set to the request code for the PPI call. Valid request codes are described later.
06–07	RECOPT	A binary halfword that must be set to one of the following values: ■ 0—No recovery is requested. ■ 1—ESTAE recovery is requested. Not valid if the program is executing in cross-memory mode.

Bytes	Name	Description
08–11	RETCODE	A binary fullword that PPI sets with the return code from the requested function. For more information about possible return codes, see the function descriptions.
12–15	WORKADR	A binary fullword that must be set to the address of a 128-byte work area. This area must be provided on all calls to the PPI. It need not be preserved across calls.
16–19	ASCBADDR	A binary fullword, that is used to contain or return the ASCB address of the current address space. This field is used in various PPI calls.
16–23	SENDERID	For those calls that require it, the one- through eight-character sender ID. If the supplied ID is less than eight characters, it must be left-justified and blank-padded.
20–23	ECBADDR	A binary fullword, returned when a receiver is defined, containing the address of an ECB that PPI posts when data arrives.
20–23	BUFFQL	A binary fullword, supplied when defining or inactivating a receiver, that contains the receiver buffer queue limit. The supplied value is used to limit the number of buffers that can be queued to the receiver.
24–31	RECVERID	For those calls that require it, the one- through eight-character receiver ID. If the supplied ID is less than eight characters, it must be left-justified and blank-padded.
32–35	BUFFLEN	A binary fullword, supplied on some PPI calls, and returned on others, that contains the length of a supplied buffer or data, or the returned actual data length.
32–33	AUTHIND	A binary halfword, used when defining a receiver to indicate whether senders must be APF authorized or not. The following values are allowed: ■ 0—No APF authorization is required. ■ 1—Senders must be APF authorized to send to this receiver.
34	FLAG1	Bit 0 (X'80') is set or reset by a request code 2 (query receiver) to indicate whether the receiver queue is full. The bit is set if the queue is not full or reset if full.
36–39	BUFFADDR	A binary fullword that must be set to the address of a sending or receiving buffer on some calls. The length of this buffer must be set in the BUFFLEN field.

Bytes	Name	Description
36–39	TCBADDR	A binary fullword that must be set to the address of the current TCB when defining a receiver, or is returned on the 'get TCB/ASCB addrs' call.
40–45	RESERVED	These fields are not used in z/OS.
46	REQIND1	Request indicator 1. If you do not want to change an existing receiver definition (even when the TCB/ASCB addresses match), set bit 0 (X'80') before issuing a DEFINE RECEIVER (request code 4). A return code of 16 is returned instead. If you want to determine the version of PPI in use, set bit 1 (X'40') before issuing a QUERY PPI STATUS (request code 1).
47	REQIND2	Request indicator 2 (presently unused).
48–51	VERSION	Request code 1 returns the PPI version here if requested to (by setting REQUIND1 to X'40' before the call). The version returned is a fullword 1 indicating Tivoli NetView 2.4.
52–55	RESERVED	Presently unused.

Specify Sender and Receiver IDs

The SENDERID and RECVERID fields of the RPB are used to specify sender and receiver IDs as required. These IDs must adhere to the following rules:

- 1 to 8 characters-if shorter than 8 characters, must be left justified and blank padded, embedded blanks are not allowed.
- The following characters can be used: A-Z, 0-9, @, #, $, %, &; no other characters are allowed.
- IDs starting with NETV or NETM are reserved and can only be defined by your product's main task.

Make the Call

When making the PPI call, the following conventions must be followed:

- Register 1 must point to a full word in storage that points to the RPB. The end of parmlist bit (bit 0) of the full word can, but need not, be set.
- Register 13 must point to a standard 18-word save area.
- Register 14 must contain the return address in the calling program. This is normally done using the BALR instruction.
- Register 15 must contain the entry point of CNMNETM (or CNMNETV).
- The program must be in primary addressing mode, not secondary, AR, or HOME mode. It must be in TCB mode. SRB mode callers and cross-memory callers are not supported.

In an XA or ESA environment, all addresses in these registers and in the pointer to the RPB must be valid for the AMODE of the calling program. There is no requirement to be in 31 bit mode.

Note: If a LOAD macro was used to obtain the address of CNMNETM (CNMNETV), the top bit of the returned address can be set. It is acceptable to use BALR, BASR, or BASSM to call it.

Most high-level languages follow the conventions automatically.

Upon return, registers are restored. Register 15 is set to 0. It does not contain a return code. The return code is in the RPB.

Sample PPI Calls

Some examples of PPI calls in various languages follow:

Assembler

```
CALL CNMNETV, (RPB)
```

```
CALL (R15), (RPB)
```

PL/1

```
CALL CNMNETV (RPB);
```

C

```
CNMNETV (RPB);
```

COBOL

```
CALL 'CNMNETV' USING RPB
```

Notes:

- CA recommends that CNMNETV not be link-edited with the program.
- In Assembler, use the LOAD macro to load CNMNETV and save the returned address.
- In PL/1, use the FETCH statement to load CNMNETV.
- In COBOL (VS COBOL II, pointer ability is needed), use the DYNAM option.

Check PPI Status

To check the status of PPI, use request code 1. This request code allows an application program to inquire about the status of the PPI facility. The return code indicates whether it is active or not.

The RPB fields in the following table must be set:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	1.
12–15	WORADDR	The address of a 128-byte work area.
46	REQIND1	Bit 1(X'40') if you want the PPI version returned.
48–51	VERSION	Binary 0 so that a back-level PPI can be detected correctly.

The RPB fields in the following table are returned after the call:

Bytes	Field Name	Contains
08–11	RETCODE	The return code.
48–51	VERSION	The PPI version number if RETCODE 10 is returned. F'1' indicates Tivoli NetView 2.4. F'0' indicates a prior release.

The following return codes are possible:

10

PPI is active and available to process requests.

24

PPI is not active.

28

PPI requests are not supported.

90

A processing error occurred.

Control Receiver Programs

There are a number of request codes that can be used to obtain information from, and control receiver programs:

2

Check the status of a receiver program

3

Obtain the ASCB and TCB addresses of a receiver program

4

Define and initialize a receiver program

9

Deactivate a receiver program

10

Delete an active receiver program

Each of these functions is described in the following sections.

Check the Status of a Receiver Program

To check the status of a receiver program, use request code 2. The return code from this request indicates whether the receiver is defined, active, or inactive.

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	2.
12–15	WORADDR	The address of a 128-byte work area.
24–31	RECVERID	The name of the receiver you want to query.

The RPB fields in the following table are returned after the call:

Bytes	Field Name	Contains
08–11	RETCODE	The return code.
34	FLAG	The status of the receiver queue (for RETCODE 14 and 15). Bit 0 is set if there is space on the receiver queue or cleared if the receiver queue is full.

The following return codes are possible:

14

The receiver program is active.

15

The receiver program is not active.

22

The requestor is not in primary addressing mode.

24

PPI is not active.

26

The receiver program is not defined.

28

PPI requests are not supported.

90

A processing error occurred.

Obtain ASCB and TCB Addresses

To obtain ASCB and TCB addresses, use request code 3. This request code allows application programs written in languages that do not allow you to obtain the ASCB and TCB addresses of the current program, to obtain them. These addresses are required for some other PPI request codes.

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	3.
12–15	WORADDR	The address of a 128-byte work area.

The RPB fields in the following table are returned after the call:

Bytes	Field Name	Contains
08–11	RETCODE	The return code.
16–19	ASCBADDR	The address of the current ASCB.
36–39	TCBADDR	The address of the current TCB.

The following return codes are possible:

0

Request completed successfully.

22

The requestor is not in primary addressing mode.

24

PPI is not active.

28

PPI requests are not supported.

90

A processing error occurred.

Define and Initialize a Receiver

To define or initialize a receiver, use request code 4. This request code lets you perform the following functions:

- Define a receiver—it declares the name of the receiver and the buffer limit.
- Initialize a receiver that does not exist.
- Reactivate a defined but inactive receiver.
- Alter the buffer queue limit. This function can be performed at any time, as long as all the fields are specified, and the TCB/ASCB addresses match those addresses of the defined receiver.

  Changing the value of the buffer queue limit does not drop buffers if it is reduced. The change only prevents additional buffers from being queued.

  Prevent the accidental overwrite of an existing receiver definition by setting REQIND1 to X'80'. This flag causes an exclusive-active check to be made, that is, if the receiver is presently active, no change is made to it, regardless of a match on the TCB/ASCB addresses.

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	4.
06–07	RECOPT	A recovery option as required.
12–15	WORADDR	The address of a 128-byte work area.
16–19	ASCBADDR	The address of the current ASCB (can be obtained by a request type 3).
20–23	BUFFQL	The buffer queue limit: 0 to PPI maximum parameter.
24–31	RECVERID	The receiver ID.
32–33	AUTHIND	0 if senders do not require authorization or 1 if senders must be APF authorized.
36–39	TCBADDR	The address of the current TCB (can be obtained by a request type 3).
46	REQIND1	Bit 0 (X'80') if an exclusive-active check is wanted.

The RPB fields in the following table are returned after the call:

Bytes	Field Name	Set to...
08–11	RETCODE	The return code.
20–23	ECBADDR	The address of the receiver ECB.

The following return codes are possible:

0

Request completed successfully—ECB address is set.

16

The receiver program is already active, and the TCB/ASCB address did not match, or they matched but the exclusive-active check flag (REQUIND=X'80') is set.

22

The requestor is not in primary addressing mode.

24

PPI is not active.

25

The ASCB address is not correct.

28

PPI requests are not supported.

32

No storage is available.

36

ESTAE could not be established as requested.

40

Receiver ID is invalid.

90

A processing error occurred.

Notes:

- Only this product can define receiver ID names starting with NETM or NETV, unless the PPINETM/R parameters are set to NO.
- The returned ECB-address field provides the address of an ECB, in the TCB key of the caller, in a protected subpool (that is, cannot be freed by user code), that is posted when buffers are available to the receiver. Do not alter this ECB, which is automatically cleared and posted when relevant.
- The return code is 0 if buffers are available, and 99 if the PPI facility is shutting down.
- Tivoli NetView 2.4 lets you define a receiver passing a zero ASCB address—in this case, it does not allocate an ECB. The SOLVE PPI does not support this feature and returns RETCODE=25 (invalid ASCB address).

Deactivate a Receiver

To deactivate a receiver program, use request code 9. If a receiver is not deactivated explicitly, it is deactivated at end of task or end of address space of the associated task/address space. Explicit deactivation lets you reset the buffer queue limit.

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	9.
06–07	RECOPT	A recovery option as required.
12–15	WORADDR	The address of a 128-byte work area.
16–19	ASCBADDR	The address of the current ASCB (can be obtained by a request type 3).
20–23	BUFFQL	The buffer queue limit: 0 to PPI maximum parameter.
24–31	RECVERID	The receiver ID.

The RPB field in the following table is returned after the call:

Bytes	Field Name	Set to...
08–11	RETCODE	The return code.

The following return codes are possible:

0

Request completed successfully—ECB address is set.

15

The receiver program is already inactive.

22

The requestor is not in primary addressing mode.

24

PPI is not active.

25

The ASCB address is not correct.

26

The receiver program is not defined.

28

PPI requests are not supported.

36

ESTAE could not be established as requested.

40

Receiver ID is invalid.

90

A processing error occurred.

Note: Only the owning task/address space can explicitly deactivate a receiver.

Delete an Active Receiver

To delete an active receiver program, use request code 10. Any unreceived data buffers are purged.

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	10.
06–07	RECOPT	A recovery option as required.
12–15	WORADDR	The address of a 128-byte work area.
16–19	ASCBADDR	The address of the current ASCB (can be obtained by a request type 3).
24–31	RECVERID	The receiver ID.

The RPB field in the following table is returned after the call:

Bytes	Field Name	Set to...
08-11	RETCODE	The return code.

The following return codes are possible:

0

Request completed successfully—the receiver has been deleted.

15

The receiver program is already inactive.

22

The requestor is not in primary addressing mode.

24

PPI is not active.

25

The ASCB address is not correct.

26

The receiver program is not defined.

28

PPI requests are not supported.

36

ESTAE could not be established as requested.

40

Receiver ID is invalid.

90

A processing error occurred.

Note: Only the owning task or address space can explicitly delete a receiver.

Send a Generic Alert

To send a generic alert, use request code 12. No special PPI setup is required.

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	12.
06–07	RECOPT	A recovery option as required.
12–15	WORKADDR	The address of a 128-byte work area.
32–35	BUFFLEN	The length of the generic alert data.
36–39	BUFFADDR	The address of the generic alert data.

The RPB field in the following table is returned after the call:

Bytes	Field Name	Set to...
08–11	RETCODE	The return code.

The following return codes are possible:

0

Request completed successfully.

4

The ALERT receiver task (NETVALERT) is not active—the alert has been queued.

22

The requestor is not in primary addressing mode.

24

PPI is not active.

26

The NETVALERT receiver program is not defined.

28

PPI requests are not supported.

32

No storage is available.

33

The buffer length is invalid.

35

Alert receiver buffer queue is full.

36

ESTAE could not be established as requested.

40

Sender ID is invalid.

90

A processing error occurred.

Note: The generic alert must include the 8-byte NMVT header.

The default buffer queue limit for the alert receiver is 1000 generic alerts.

A return code of 22 or greater means that the alert has not been copied to the alert receiver queue.

If no hierarchy or resource list subvector is provided in the generic alert, the sender ID is used as the resource name.

PPI does not release the data buffer storage. Your program must release storage if necessary.

Control Data Buffers

There are a number of request codes that can be used to control the sending and receiving of data buffers:

14

Sending a data buffer to a receiver program

22

Allowing a receiver program to receive a data buffer

23

Purging a data buffer from a receiver program

Send a Data Buffer to a Receiver

To send a data buffer to a receiver program, use request code 14. The receiver program does not need to be active. As long as its buffer queue limit is not exceeded, the data buffer is queued to it.

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	14.
06–07	RECOPT	A recovery option as required.
12–15	WORKADDR	The address of a 128-byte work area.
16–23	SENDERID	A valid sender ID.
24–31	RECVERID	The target receiver ID.
32–35	BUFFLEN	The length of the data buffer to send.
36–39	BUFFADDR	The address of the data buffer to send.

The RPB field in the following table is returned after the call:

Bytes	Field Name	Set to...
08–11	RETCODE	The return code.

The following return codes are possible:

0

Request completed successfully.

4

The specified receiver is not active-the buffer has been queued.

22

The requestor is not in primary addressing mode.

23

The sender program is not authorized.

24

PPI is not active.

26

The receiver program is not defined.

28

PPI requests are not supported.

32

No storage is available.

33

Buffer length is invalid.

35

The receiver buffer queue is full.

36

ESTAE could not be established as requested.

40

Sender ID is invalid.

90

A processing error occurred.

Notes:

- If sending a buffer to a receiver defined as authorized, the sender must be APF authorized.
- After the call the data buffer is queued to the nominated receiver.
- The PPI does not release the data buffer storage. Your program must release the storage if necessary.

Allow a Receiver to Receive a Data Buffer

To allow a defined, active receiver to receive a data buffer, use request code 22. The next buffer in the receiver buffer queue is made available in the user-provided area.

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	22.
06–07	RECOPT	A recovery option as required.
12–15	WORKADDR	The address of a 128-byte work area.
16–19	ASCBADDR	The receiver ASCB address.
20–23	ECBADDR	The receiver ECB address—this field is set only when RETCODE 30 is returned.
24–31	RECVERID	The target receiver ID.
32–35	BUFFLEN	The length of the data buffer to send.
36–39	BUFFADDR	The address of the data buffer to send.

The RPB fields in the following table are returned after the call:

Bytes	Field Name	Set to...
08–11	RETCODE	The return code.
16–23	SENDERID	The ID of the sender of this buffer.
32–35	BUFFLEN	The actual data buffer length.

The following return codes are possible:

0

Request completed successfully.

22

The requestor is not in primary addressing mode.

24

PPI is not active.

25

The ASCB address is not correct.

26

The receiver program is not defined.

28

PPI requests are not supported.

30

No data buffers are in the receiver buffer queue.

31

The receiver buffer size is not large enough for the incoming data buffer.

33

Buffer length is invalid.

35

The receiver buffer queue is full.

36

ESTAE could not be established as requested.

40

Sender ID is invalid.

90

A processing error occurred.

The correct ASCB address is required. This address is the ASCB address provided when the receiver was defined.

One data buffer at a time can be received. They are provided first-in-first-out order. If the call is successful, the sender ID of the sending program is provided.

The length of the incoming buffer is provided in the BUFFLEN field after the call. If a return code 31 (buffer too small) is given, BUFFLEN contains the required length to receive the buffer successfully. The call can be reissued after obtaining a large enough buffer.

If no data is queued, a return code of 30 is given and the ECBADDR field is set to the ECB address, as returned by DEFINE RECEIVER. The receiver can wait (using the WAIT macro or request code 24) until more data arrives.

Purge the Data Buffer

To allow the caller to purge the front buffer on a receiver queue, use request code 23. It is equivalent to receiving the buffer and ignoring it, except that no receiver buffer is needed, which is useful for purging buffers that are too long.

The RPB fields in the following table must be set up before the call:

Bytes	**Field Name**	**Set to...**
00–03	RPLEN	56.
04–05	REQUEST	23.
06–07	RECOPT	A recovery option as required.
12–15	WORKADDR	The address of a 128-byte work area.
16–19	ASCBADDR	The address of the current ASCB.
24–31	RECVERID	The target receiver ID.

The RPB field in the following table is returned after the call:

Bytes	**Field Name**	**Set to...**
08–11	RETCODE	The return code.

The following return codes are possible:

0

Request completed successfully—the front data buffer has been purged.

22

The requestor is not in primary addressing mode.

24

PPI is not active.

25

The ASCB address is not correct.

26

The receiver program is not defined.

28

PPI requests are not supported.

30

No data buffers are in the receiver buffer queue.

36

ESTAE could not be established as requested.

40

Sender ID is invalid.

90

A processing error occurred.

Notes:

- The correct ASCB address is required-that is, the ASCB address provided when the receiver was defined.
- Only one data buffer at a time can be purged.
- If no buffers are queued, return code 30 is returned.

Wait on an ECB

To allow the caller to wait on an ECB, use request code 24. This request code allows receiver programs written in languages that do not support a WAIT service to wait for input.

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	24.
12–15	WORKADDR	The address of a 128-byte work area.
20–23	ECBADDR	The address of the ECB.

The RPB field in the following table is returned after the call:

Bytes	Field Name	Set to...
08–11	RETCODE	The return code.

The following return codes are possible:

0

Request completed successfully—the ECB has been posted.

18

The ECB was not 0 on entry to this function—it might have been already posted.

22

The requestor is not in primary addressing mode.

24

PPI is not active.

28

PPI requests are not supported.

Important! Use this function only if your programming language does not support a WAIT facility. Unpredictable results can occur if the PPI subsystem is terminated while you are waiting using this request.

The ECB address is returned by a request code 4 (define a receiver).

You can wait using request code 24 after receiving a return code 30 from receive.

Remember to test the ECB post code in the last byte of the ECB for 0 (data available) or 99 (PPI shutting down).

Obtain a Unique Sender or Receiver ID

To obtain a unique sender or receiver ID, use request code 60. This request code is useful when establishing a bidirectional conversation with another program. The other program can be a globally known program, with a known ID. However, this program cannot use a reserved, unique ID. In this case it can use this service to obtain a valid, unique ID.

This service is only supported by this product's implementation of PPI (known as SOLVE PPI).

The RPB fields in the following table must be set up before the call:

Bytes	Field Name	Set to...
00–03	RPLEN	56.
04–05	REQUEST	24.
12–15	WORKADDR	The address of a 128-byte work area.

The RPB fields in the following table are returned after the call:

Bytes	Field Name	Set to...
08–11	RETCODE	The return code.
24–31	RECVERID	The returned unique ID.

The following return codes are possible:

0

Request completed successfully—a unique ID has been allocated.

20

Function is invalid (not the SOLVE PPI).

22

The requestor is not in primary addressing mode.

24

PPI is not active.

28

PPI requests are not supported.

90

A processing error has occurred—this error also indicates that no more unique IDs are available.

The returned ID is in the format: *pppp*&*nnn* where *pppp* is the 1- to 4-character PPI name prefix, as set by one of the PPI SSI JCL parameters. If less than four characters, it is padded to four characters with ampersand (&) characters. Position 5 is always an ampersand. Positions 6 through 8 are allocated from the following characters: 0 through 9, A through Z, @, #, $, and %. This allocation allows up to 64000 unique IDs.

Unique IDs can be reused. If an ID is obtained, defined and inactivated, and nothing is queued to it, then, subject to the PPIINATO and PPIREUSE SSI startup parameter values, the ID can be made available for reuse.

If the PPI prefix is set to the same value as the domain ID associated with the product, then these IDs are also unique across the connected network.

Receive Information from a Receiver Program

When you receive information from a receiver program after sending a PPI call, a number of return codes are possible. The following table provides a full list of possible PPI return codes, a complete description of the possible causes, and all request codes that can cause that return code.

Note: These are the RPB return codes. NCL interface return codes (in &RETCODE) are different. The NCL interface returns the PPI return code in &ZFDBK for information purposes.

Return Code	Description	Returned on Requests
0	The PPI request has completed successfully. The requested function has been performed with no errors or warnings.	3, 4, 9, 12, 14, 22, 24, and 60
4	The specified receiver is not presently active. The data buffer or generic alert has been queued. (The receiver buffer queue was not full).	12 and 14
10	The PPI facility is active and can be used. This return code is good.	1
14	The receiver program is active.	2
15	The receiver program is (already) inactive. The program is defined, but has been explicitly or implicitly deactivated.	2 and 9
16	The receiver program is already active. The program cannot be defined.	4
18	The ECB is not 0 (normally the receiver ECB). This condition is not necessarily an error because the ECB can be posted after it was last checked.	24

Return Code	Description	Returned on Requests
20	Request type is invalid. The passed request code was not valid. This condition can occur with request code 60 if this product is not providing the PPI. Otherwise, it is probably an error in formatting of the RPB.	Any incorrect request
22	The program is not executing in primary addressing mode. Programs must not call the PPI in secondary mode (z/OS). They must be in primary mode, although they can be in cross-memory mode (that is primary ^= home).	1, 2, 3, 4, 9, 12, 14, 22, 24, and 60
23	The program is not authorized. Attempt to send a buffer to a receiver defined as authorized.	14
24	PPI is not active. The SSI is not active or PPI is not running on a SOLVE SSI or Tivoli NetView SSI. In this product's implementation, can also indicate that this system does not support PPI, as it has not built the relevant control block structure.	All
25	The ASCB address is not correct. To deactivate a receiver or receive data, the correct ASCB address must be supplied.	9 and 22
26	The receiver is not defined. The supplied receiver name has never been defined to this execution of PPI. **Note:** This condition could occur if the PPI-owning SSI is stopped and restarted.	2, 9, 12, 14, and 22
28	This product does not support PPI. This return code cannot occur with the implementation of the SOLVE PPI. It can occur if Tivoli NetView is providing PPI services.	All
30	No data buffers are available in the receiver queue. There is no data to be received. PPI clears the receiver ECB automatically.	22
31	The receiver data buffer length is too short to receive the next data buffer. The RPB buffer length contains the length of the pending buffer.	22
32	No storage available. Unable to obtain storage required to complete the requested function.	4, 12, and 14
33	Invalid buffer length. The supplied RPB or data buffer length was not valid. For a data buffer, the length must be less than 1 or greater than the PPI maximum buffer size JCL parameter.	All

Return Code	Description	Returned on Requests
35	The receiver buffer queue is full. As the queue is full, the new data buffer is not added to the queue.	12 and 14
36	Unable to establish ESTAE protection as requested. Either the caller is not in home mode, or the ESTAE gave a nonzero return code.	All
40	The specified receiver or sender ID is invalid. The value does not satisfy the documented rules for these IDs, or (when defining, deactivating, or receiving for a receiver), starts with NETV or NETM and the requesting task is not the SSI-connected product region.	4, 9, 12, 14, and 22
90	A processing error has occurred. This return code is a catch-all that covers many things: ■ ESTAE-trapped ABENDs ■ Various internal errors	All

In addition to these return codes, some fundamental errors can cause S0C4 ABENDs. These errors would include an RPB that cannot be referenced, or a bad work-storage address. A bad save area address (for example, a 31-bit address for a 24-bit mode caller) can also cause it.

Trace the Cause of a Processing Error

Return code 90 indicates that a processing error occurred. This return code has a trace facility that lets you determine the cause of the processing error. This facility is enabled during SOLVE SSI setup.

When you issue a PPI call, processing messages from the receiver program are written as WTOs to your system console. If tracing is enabled, and a return code of 90 is recorded in register 15, register 0 records debugging information which is also sent to the system console. This additional information can be used by you to debug the processing error.

Note: For more information about PPI parameters, see the *SOLVE Subsystem Interface Guide*.

Register 0 contains the debugging information as follows:

```
X'rrrrmm5A'
```

rrrr

Is the reason code of the debugging information, in signed hexadecimal, length 2 bytes. For example, a reason code of 10 would be X'000A' and -141 would be X'FF73'.

mm

Is the module identification of the debugging information in hexadecimal. For example, 35 would be X'23'.

5A

Is 90 in hexadecimal, which indicates that the debugging information has been set in R0.

Following is the resulting message format that you receive on the system console:

```
NS3580 PPI RC=90 MOD=aa REAS=bbbbbb - RPB FOLLOWS...
NS3581 RPB +nnn xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
NS3581 ... (for total RPB length)
NS3582 *END*
```

The fields in the messages are:

MOD=aa

Contains the module identification of the debugging information, in decimal. For example, MOD=35.

REAS=bbbbbb

Contains the (signed) reason code of the debugging information, in decimal. For example, REAS=-00121.

+nnn

Contains the offset, in hexadecimal, of the start of the hexadecimal dump of the RPB (up to 16 bytes is dumped per line).

xxxxxxxx

8 hexadecimal digits representing RPB data.

Debugging Codes

The following table provides a list of all the most likely combinations of debugging module IDs and reason codes, with a brief description of the causes of each. If you receive any other module IDs, contact the help desk.

MOD	REASON	R0	Description
35	1	0001235A	No work storage address or bad address
35	2	0002235A	Recovery option not 0 or 1
35	4	0004235A	ESTAE exit driven
36	4	0004245A	ESTAE exit driven
36	5	0005245A	Bad BUFQ-LIM value
36	6	0006245A	Bad RPB AUTH-IND option value (not 0 or 1)
37	1	0001255A	Define receiver TCB/ASCB not correct
37	2	0002255A	Redefine inactive receiver TCB/ASCB incorrect
37	4	0004255A	FRR driven during Send Buffer processing
37	5	0005255A	FRR driven during Receive Buffer processing
37	6	0006255A	All unique names in use (PPI func. 60)

Chapter 24: &NDB Verbs, Built-in Functions, and System Variables

This section provides a summary of verbs, built-in functions, and system variables used by NetMaster databases (NDBs).

Note: For more information about these verbs, see the *Network Control Language Reference Guide*.

This section contains the following topics:

&NDB Verb Summary

&NDBADD	Adds a new record to an NDB.
&NDBCTL	Sets NDB NCL processing options.
&NDBCLOSE	Signs off (disconnect) from an NDB.
&NDBDEF	Adds or deletes field definitions to/from an NDB.
&NDBDEL	Deletes a record from an NDB.
&NDBFMT	Defines a data format list for retrieval of data from an NDB.
&NDBGET	Retrieves a record from an NDB.
&NDBINFO	Retrieves information about an NDB or field definitions in the NDB.
&NDBOPEN	Signs on (connect) to an NDB.
&NDBSCAN	Finds a set of records in an NDB that matches a set of search criteria.
&NDBSEQ	Defines a sequential retrieval path into an NDB.
&NDBUPD	Updates a record in an NDB.

Built-in Function Summary

&NDBPHON

The &NDBPHON function is a phonetic conversion function which allows conversion of a character string into a phonetic key. The conversion algorithm supplied is SOUNDEX. However, a user-supplied exit can be called. The user exit can implement any approach to phonetic conversion.

The SOUNDEX algorithm is as per KNUTH (Art of Computer Programming, Volume III). The returned value is Knnn where K is the first character and nnn is the SOUNDEX coded value.

A sample exit, PHONEX01, shows how to write a phonetic exit. The NDBPHONX SYSPARMS supply the name of the user exit.

&NDBQUOTE

Allows an NCL procedure to automatically quote data for the &NDBADD, &NDBGET, &NDBSCAN, and &NDBUPD statements.

System Variable Summary

&NDBERRI

Provides extra information about some errors.

&NDBRC

Provides a return code indicating success or otherwise after an &NDB statement.

&NDBRID

Provides the current record ID after some &NDB statements.

&NDBSQPOS

Indicates the position of a returned record in a sequence built by &NDBSCAN.

Free-form Syntax

Several NCL statements use a special syntax, different from normal NCL syntax, to allow easy coding of data definitions and scan requests. The relevant statement descriptions indicate the part of the statement that uses the free-form syntax. The free-form part must always be coded after any fixed-form, standard NCL-syntax parameters on the same statement. The rules for this free-form syntax are:

- Blanks are only required to delimit adjacent words, except inside data values. Blanks are not required, but may be specified, around or next to special characters that act as delimiters.

  Blanks inside data values are significant, except that trailing blanks are never stored in character-format data.

  NCL variables with blanks in the value are regarded as a special case, and the blank is regarded as part of the data value. This is because blanks inside NCL variables are represented internally in a special way.

- The following special characters act as delimiters, unless enclosed in a quoted string. They have special meaning to the syntax:

  (Left bracket

) Right bracket

  : Colon (meaning: range)

  = Equal sign

  ¬ Not sign

  < Less than sign

  > Greater than sign

  & Ampersand (meaning: AND)

  | Bar (meaning: OR)

  , Comma

  Real blank (not embedded in an NCL variable)

  Certain combinations of these characters are treated as a single token for parsing. These combinations are p =, < =, >=, and =<, =>, meaning not equal, less than or equal, greater than or equal, less than or equal, and greater than or equal, respectively.

- Values may be enclosed in quoted strings whenever the value contains a special character, or a real blank.

  The quotes can be single (') or double(”). If the data value being quoted contains a single or double quote, you can quote the data with the other quote, or double up each occurrence of the quote character.

  For example, 'It”s a quoted value' will be regarded as the value It's a quoted value.

  The &NDBQUOTE built-in function provides an easy way to automatically quote data when necessary.

  A data value can always be quoted, even date, hexadecimal, or numeric values. Quoting also prevents any possibility of the value being regarded as a keyword.

- The following words can be used instead of special characters, as an aid to clarity. If surrounded by other words, ensure at least one blank separates them.
  - EQ can be used to replace =
  - NE can be used to replace ¬=
  - LT can be used to replace<
  - GT can be used to replace >
  - LE can be used to replace=<
  - GE can be used to replace=>
  - AND can be used to replace&
  - OR can be used to replace |
  - TO can be used to replace :
  - NOT can be used to replace ¬

- Several statements support a START/DATA/END construct to allow free-form expressions to be constructed that are longer than a single NCL statement is allowed to be. These statements can be coded as:

  ```
  &NDBxxxx  dbname [ parameters ] [ DATA ] free-form text
  ```

  if the free-form text can fit on one statement (with possible continuations).

  To overcome NCL statement length limitations, and also to allow the free-form text to be built piece-meal (for example, by indirect variable reference), the statements can also be coded as:

  ```
  &NDBxxxx dbname [ parameters ] START
  &NDBxxxx dbname [ DATA ]  part-of-free-form-text
  &NDBxxxx dbname END
  ```

  The free-form text can be broken anywhere a blank is valid. Any number of intermediate statements can be used to build the complete free-form text. The database is not called until the END statement is encountered.

  Any other parameters must be coded on the &NDBxxx START statement.

  Note: The statements may be interspersed with other NCL statements, including statements referencing other or even the same database, and even statements building free-form text for the same database, as long as they are different statements. That is, you can be concurrently building a multi-statement add and update for the same database, but not two different adds for the same database.

  To cancel a partially built statement, use:

  ```
  &NDBxxx dbname CANCEL
  ```

  This statement is valid even if no current START/END set is being built; thus it can be used in general error routines.

Appendix A: NCL VSAM Techniques

This section contains the following topics:

Initialization and ACB Open Processing

You can allocate and open UDBs at any time by using the ALLOCATE and UDBCTL commands. With certain UDBs, you might want to use some of the more advanced VSAM facilities that your product supports. For example, you might want to specify use of the LSR pool or sequential insert strategy (SIS) to improve system performance.

Use of these optional facilities must be requested on the UDBCTL command at open time. During the open process, a VSAM ACB is generated for each data set and an open attempted. If the open fails, the system internally links to IDCAMS to perform a verify of the data set. Therefore, it is not necessary to include verify steps in the system JCL.

Automatic Verification and Loading

On completing the verify function, the open is re-attempted. If the data set is empty, the system closes and re-opens the data set in create mode and attempts to perform a load according to the following rules:

For Entry-Sequence Data Sets (ESDS)

For an ESDS a single record in the following format is loaded:

```
N28510 VSAM INITIAL LOAD PERFORMED AT hh.mm.ss
ON day-dd.mon.year
```

This initial load record remains in the ESDS and if necessary can be skipped during offline processing by using the IDCAMS utility to REPRO the data set specifying the SKIP=1 operand.

Note: This load process is performed only if the data set is classified by VSAM as being empty. This implies that the high-order RBA is 0. If data exists within the data set when it is opened, then the load process is bypassed.

In a z/OS environment, where output is sent directly to JES2 or JES3, no load processing is required and hence the N28510 message is not the first record in the file.

If the load fails, the data set is classified as unusable and is blocked from further processing until the problem is corrected. The SHOW UDB command can be used to determine the current status of a UDB and displays any error code detected during open processing. If necessary, the data set in a z/OS environment can be de-allocated using the DEALLOC command to assist in offline correction of problems.

For Key-Sequenced Data Sets (KSDS)

For a KSDS, a single record with a key of all X'00' is inserted into the data set. If the load is successful, the data set is closed and re-opened in update mode and the initial load record deleted. If the load fails, the data set is classified as unusable and is blocked from further processing. The SHOW UDB command can be used at any time to determine the status of a UDB and if necessary to obtain any error code set during open processing.

The loading of alternate indices by your product is not supported. This must be performed using the IDCAMS BLDINDEX function. Once built, these indices are maintained correctly.

RPL Handling

Until a file is opened (using an &FILE OPEN statement) by an NCL procedure, no RPLs are created and no work buffers allocated.

Before a UDB can be referenced by a procedure, the UDBCTL command must be used to assign a logical file ID to the physical data set. This command can be included in the NMINIT procedure or entered from OCS.

To open a file, an NCL procedure uses an &FILE OPEN statement. This statement references the logical file ID previously assigned by the UDBCTL command, and in doing so completes the link between the NCL procedure and the actual data set.

It is at this time that a VSAM RPL is created for use by the NCL procedure. This RPL is used in all subsequent requests for this data set by the procedure. The RPL remains in existence until either explicitly freed by the &FILE CLOSE statement, or termination of the NCL procedure. When processing with multiple files, one RPL is created for each new file ID the first time that the file is opened.

Obtain I/O Buffers

At the time the RPL is generated an I/O buffer is also obtained. This buffer is large enough to hold the largest record that could be read from, or written to, the UDB. Therefore, it is important to accurately define the record sizes when the VSAM cluster is allocated. Specification of record sizes in excess of that required not only impacts VSAM algorithms for space allocation, but also forces your product to obtain buffers of an unnecessary size, thus wasting storage.

Concurrent Access to Multiple UDBs

An NCL procedure can actively process several UDBs at a time. The mandatory ID= operand on the &FILE verbs, indicates which UDB the verb is to be actioned upon. Each file must be opened separately, using an &FILE OPEN statement before reads and writes can be performed on it. There is no overhead in processing several files simultaneously, because NCL merely swaps pointers between the work buffers and RPLs used for each file, according to the ID specified on the &FILE verb.

Current keys and data set positioning are remembered by NCL when processing swaps from one file to another, so it is not necessary for the NCL procedure to remember these.

The example in the next section, of copying one file to another, shows how processing can be alternated between two files.

Data Set Positioning and Generic Retrieval

NCL supports both sequential and generic retrieval from keyed data sets. Such functions imply that a current position within the file is maintained. Thus, the NCL procedure can simply request the next record and it is supplied.

Example: Data Set Positioning and Generic Retrieval

```
&FILE OPEN ID=FILE1 FORMAT=DELIMITED   -* Open file 1
&FILE OPEN ID=FILE2 FORMAT=DELIMITED   -* Open file 2
.LOOP
   &FILE GET ID=FILE1 OPT=SEQ VARS=A* RANGE=(1,6)
                                       -* Read record
   &IF &FILERC NE 0 &THEN &GOTO .ENDPROC
                                       -* End if not 0
   &X = &FILEKEY                       -* Copy key of
                                       -* record that was
                                       -* read into
                                       -* variable
   &FILE PUT ID=FILE2 KEYVAR=X VARS=(A1,A2,A3,A4,A5,A6)
                                       -* Write record
   &GOTO .LOOP                         -* Loop to read
                                       -* next record on
                                       -* file 1
```

Note: This example does not fully cater for such things as error conditions.

It is not necessary for the NCL procedure to increment keys.

Under certain circumstances, such as with generic retrieval, it might be necessary to alter the retrieval sequence and commence retrieval using a different key.

NCL must be informed that such a change is required and that the current retrieval sequence is to be stopped. This is done using the &FILE GET ID=fileid OPT=END statement. This indicates to NCL that generic retrieval is to be terminated in anticipation of some other processing.

If an end-of-file condition is signaled, no &FILE GET ID=fileid OPT=END is required. The use of a non-generic function, such as the specific retrieval of a record, also cancels a previous generic function.

Release File Processing Resources

The &FILE OPEN statement allocates certain resources to the requesting NCL procedure. It is not normally necessary to release file processing resources within an NCL procedure. This is performed automatically when the NCL procedure terminates.

Under certain circumstances, such as in an EASINET procedure, where there can be many concurrent users performing file processing, it might be desirable to release any file processing overheads when they are no longer required, to ensure that system overheads are minimized.

This can be done with the &FILE CLOSE statement. &FILE CLOSE allows either specific files or all files to be freed. When this is done, any storage associated with processing the file is released and the connection is logically severed for that user.

Having used &FILE CLOSE to release a particular file, connection can be re-established using another &FILE OPEN statement.

&FILE CLOSE destroys any generic retrieval position a user might have established within a file and any subsequent reference would have to re-establish that position if required.

Display File Information

The SHOW UDB command can be used to display details about files available to the system. This information includes details about the number of active users, space usage, and the status. In addition, any open error codes that caused a file to be disabled are displayed.

The SHOW VSAM command can be used to obtain additional details about the system's VSAM files. These include record and Control Interval sizes, statistics on the number of CI and CA splits and details of any buffer or string shortages that have been experienced. In OS/VS systems, this display also provides details on the performance of the Local Shared Resource (LSR) pool if one is in use.

Note: For more information about the SHOW UDB and SHOW VSAM commands and their use, see the online help.

Control UDB Performance

The techniques used by NCL should ensure efficient processing of VSAM files. Additional performance gains can be obtained by the allocation of additional buffers and processing strings.

This is achieved using the JCL AMP statement sub parameters on the DD statement for the file, or by specification of these options on the UDBCTL command:

BUFNI

The number of index buffers to be allocated by VSAM

BUFND

The number of data buffers to be allocated by VSAM

STRNO

The maximum number of concurrent strings to be used by VSAM

Additional facilities are offered as options on the UDBCTL command. They include:

SIS

Use Sequential Insert Strategy

DEFER

Use deferred writes

LSR

Use Local Shared Resource pool

Important! These parameters should only be changed if the impact on VSAM processing is clearly understood. Inadvertent changes can impose severe storage overheads which could impact the operation of other system components.

Offline Processing of Data Sets

In a z/OS environment, the DEALLOCATE command can be used to release a UDB for offline processing. Conversely, the ALLOCATE command can be used to bring a UDB on-line for processing.

In non-z/OS environments, the stripping of files, created by NCL for further offline processing, can be achieved without a restart of the system if the following rules are followed:

- VSAM SHAREOPTIONS must allow concurrent access to the data set.
- DISP=SHR must be specified on the data set.

- The UDBCTL CLOSE command is used to stop further logical connections to the file and to physically close the file. This is only successful if there are no active users currently referencing the file. The SHOW UDBUSER command can be used to display the current user of a UDB.
- Use a utility to strip the file (for example, the VSAM IDCAMS utility REPRO facility).

If the UDB has been processed using standard format, any offline processing programs must take the high-value (X'FF') field separators into account when determining the format of data within records.

To allow non-UDB format files to be processed, your product provides a format operand on the &FILE OPEN and &FILE SET statements, that designates the format of the files being processed.

Appendix B: System Level Procedures: Message Profiles

This section contains the following topics:

Use NCL Verbs to Retrieve Messages

There is a series of NCL verbs that are used to retrieve messages that are queued to particular NCL process environments. The verbs of this type are:

- &INTREAD
- &LOGREAD
- &MSGREAD
- &PPOREAD

In addition, there is the &AOMREAD verb if you have the AOM feature and &CNMREAD if you have the NEWS feature on your system.

System Level Procedure Environments

The term system level procedure applies to those specific NCL procedures that have access to specialized flows of information in a product region. In this region, there are several system level procedures, the principal ones being LOGPROC and PPOPROC. LOGPROC has access to and control over the flow of messages to the activity log. PPOPROC receives unsolicited messages from VTAM about events within the network.

The significant aspect of the special procedures is that there is only one of each in the system, and they have particular privileges and responsibilities that do not apply to the usual NCL procedures executed by product users.

MSGPROC Viewed as a System Level Procedure

Each OCS window can have a MSGPROC procedure associated with it. There is only one MSGPROC per window and it has the ability to review and process every message sent to its associated OCS window from any source, before the messages are actually delivered to the window for display.

Just as LOGPROC is the only procedure in the system that can review the messages flowing to the activity log, so an OCS window's MSGPROC is the only NCL procedure that can review and process the message traffic flowing to that OCS window.

In this respect a MSGPROC is classified as a system level procedure, even though there can be many MSGPROC procedures active in a region on behalf of many different OCS windows; MSGPROC has access to and control over a specific message flow and has privileges not open to the usual NCL procedures executed by users.

&INTREAD: Dependent Processing Environment

The last form of privileged access to traffic flow within a region occurs when &INTCMD and &INTREAD statements are used to execute commands or other NCL processes within an NCL dependent processing environment.

When a procedure executes an &INTCMD statement, command results are returned to that procedure. These results are queued and can then be read back by the original procedure through the &INTREAD statement.

&INTREAD therefore provides privileged access to the flow of command result messages produced by the various commands and NCL procedures that execute within the original procedure's dependent processing environment.

&INTREAD is also able to access unsolicited messages, such as monitor messages, for which the independent environment is profiled.

Message Handling and Processing by System Level Procedures

All system level procedures that have privileged access to a particular message flow have the following capabilities:

- They can read the next message from the traffic flow by using a privileged NCL verb statement which provides access to the queue of messages. The following verbs are used by specific system level procedures:

  &LOGREAD

  Used by the LOGPROC procedure to get a copy of the next message that is to be written to the activity log.

&PPOREAD

Used by the PPOPROC procedure to get a copy of the next message received from VTAM.

&MSGREAD

Used by a MSGPROC procedure to read the next message queued for display on the OCS window with which the procedure is associated.

&INTREAD

Used by any NCL procedure to read the next request or response queued to it via its dependent processing environment.

- They can indicate what is to be done with the message that they have received as a result of executing the 'read' statement. Typically, once a message has been received, the procedure can indicate that the message is to be processed normally (for example, &LOGCONT, &PPOCONT, &MSGCONT), or deleted and not propagated any further (for example, &LOGDEL, &MSGDEL), or replaced with an alternative message (for example, &MSGREPL).

Decide What to Do with a Message

While the system level procedures have verbs for accessing the message flow that they are to monitor and verbs for indicating the course of action to be taken with a message when it has been read, the procedures must also contain the logic necessary to analyze the messages that are read to determine what action is required.

The logic of these procedures is usually built on the concept of filtering the message flow, searching for messages that are defined as being of interest based on arguments such as message numbers or certain values that occur within the message.

In addition to processing options based on message text content, messages can also have a large number of other attributes that are not textual in nature. For example, a message can have a color attribute that causes it to be displayed in red when it is sent to a terminal.

Other messages can have the non-roll delete attribute, meaning that they will not roll off an OCS window display until specifically deleted.

To assist with the logic necessary to process the text and non-text attributes of each message, when a message is read by one of the system level procedure verbs listed earlier a message profile is created for analysis and interrogation by the procedure.

Message Profile

The concept of the message profile is that all the attributes of a message should be available as specific settings of a special group of system variables. The profile variables can be tested for specific values to assist the procedure in deciding whether special processing is needed for a message or whether the message is of no importance to the procedure. Since testing variables for specific values is easier than scanning text strings, examination of a message's profile provides a simpler and more efficient method of message analysis.

All profile variables are available in Mapped Data Object (MDO) format for the &INTREAD, &LOGREAD and &MSGREAD verbs. The object stem name for each verb is unique, but they are all mapped by the same map, that is $MSG.

- For &INTREAD the stem name is $INT.
- For &LOGREAD the stem name is $LOG.
- For &MSGREAD the stem name is $MSG.

The $MSG MDO can contain additional information that is not available in the profile variables.

The $MSG map definition is part of the your product's distributed system and can be reviewed from Mapping Services (option D.M from the Primary Menu).

Note: Most of the data components are optional.

Message Profile Variables

Following execution of any of the &INTREAD, &LOGREAD, &PPOREAD, or &MSGREAD verbs, message profile variables are set to reflect the message profile of the message read by the verb statement. The particular variables that are set depend on the verb executed and on the class of message received.

This section describes the different values that can be set in the variables that apply to the &INTREAD, &LOGREAD, and &MSGREAD verbs, and the conditions under which each of the variables can be set. The individual variables that form the message profiles of the message flows handled by the different verbs are then cross-referenced against the individual verbs in the remaining sections.

Note: System profile variables are available immediately after the &xxxREAD statement is executed and are generally available until another NCL statement is executed that alters the profile. System variables are valid only within the scope of the procedure and their values are unpredictable when control is passed to another procedure level. MDOs can be shared between called procedures using SHRVARS and explicitly on the &return statement.

Following is a list of the complete set of message profile variables that apply to these three verbs. Their MDO equivalents are included in brackets, where relevant.

Note: stem can be $MSG, $INT or $LOG depending on which verb caused the variable to be set.

&ZINTYPE

Specifies whether an &INTREAD operation has been satisfied by a request message or a response message. Values are REQ (request message) or RESP (response message) or NONE (no message).

&ZMALARM (stem.MSGATTR.ALARM)

Indicates whether the message will cause the terminal alarm to sound. Value is YES or NO.

&ZMALLMSG

Indicates whether the message was generated by a MSG ALL command. Value is YES or NO.

&ZMCOLOUR (stem.MSGATTR.DISPLAY.COLOUR)

Indicates the color attribute of the message. Value can be any one of NONE, RED, BLUE, GREEN, YELLOW, TURQUOISE, PINK, or WHITE.

&ZMDOM

Indicates whether the message is a delete operator message instruction. Value is YES or NO. This value is dependent also on the setting of the &ZMTYPE variable.

&ZMDOMID (stem.DOMID)

Contains the delete operator message identifier (domid) of the message read, if the message has the non-roll delete message attribute, as determined by the setting of the &ZMNRD variable. If &ZMNRD=NO, this variable is set to null.

&ZMEVONID (stem.SOURCE.NCLID)

Is set when the incoming message was generated by an &EVENT verb and contains the nclid of the procedure which issued the &EVENT.

&ZMEVPROF (stem.SOURCE.EVENTPROFILE)

Is set for incoming event messages (N00102) and represents the EDS profile name which resulted in delivery of the event notification.

&ZMEVRCDE (stem.EVENT.ROUTECDE)

Contains the routecde of the incoming event message (N00102) if the ROUTECDE operand was specified on the originating &EVENT verb.

&ZMEVTIME (stem.SOURCE.TIME.HHMMSSTTT)

Is set for incoming event messages (N00102) to the time the event originated. Time is in the format HH.MM.SS.TTT.

&ZMEVUSER (stem.SOURCE.USER)

Is set when the incoming message was generated by an &EVENT verb and contains the user ID of the user who issued the &EVENT verb. This variable is also set for some system events and represents the user who was responsible for the event generation.

&ZMFTSMSG

Indicates whether the message originated from File Transmission Services (FTS). Value is YES or NO.

&ZMHLIGHT (stem.MSGATTR.DISPLAY.HLITE)

Indicates the display highlighting attributes of the message. Values are NONE, USCORE, REVERSE, or BLINK.

&ZMINTENS (stem.MSGATTR.DISPLAY.INTENS)

Indicates the display intensity attribute of the message. Values are HIGH or LOW or null if no message is being processed.

&ZMLNODE (stem.SOURCE.REGION)

Indicates the terminal name of the user to whom a log message is to be attributed. Value is the name of a terminal. (This is available to &LOGREAD only.)

&ZMLOGCMD

Indicates whether a log message is an echo to the log of a command. Value is YES or NO. (This is available to &LOGREAD only.)

&ZMLSRCID (stem.PREFIX.LASTMSGID.ID)

Contains the message prefix of the last handler of the message just received.

&ZMLSRCTP (stem.PREFIX.LASTMSGID.TYPE)

Indicates the type of the last handler of the message just received. Values can be:

null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

&ZMLTIME

Contains the timestamp of a log message. (This is available to &LOGREAD only.) Time is in the format HH.MM.SS.TTT.

&ZMLUSER (stem.SOURCE.USER)

Contains the user ID to whom generation of the log message is to be attributed. (This is available to &LOGREAD only.)

&ZMAPNAME (stem.MAPNAME)

Contains the mapname of the object in $MSG.USERMDO for the current message (if present).

&ZMMONMSG

Indicates whether the message received is a monitor class message. Values are YES or NO.

&ZMMSG

Indicates whether the message received is a standard message or not. The setting of this variable is always opposite to the setting of the &ZMDOM variable and is dependent on the setting of the &ZMTYPE variable.

&ZMMSGCD (stem.MSGATTR.MSGCODE)

Indicates the (hexadecimal) message code attribute of this message. Value can be 00-FF. The message code dictates which user IDs are eligible to receive the message.

&ZMNMDIDL (stem.SOURCE.LAST.DOMAIN)

The domain ID of the last region to handle this message. This can be the same as the originating system, or different if the message was originated by a remote system and then routed onwards by an intermediate system.

&ZMNMDIDO (stem.SOURCE.ORIG.DOMAIN)

The domain ID of the region from which this message originated. If sourced from the local system it will contain the local system's domain ID. If sourced from a remote system this variable carries the domain ID of the originating system even though the message might have been routed onwards by intermediate systems.

&ZMNRD

Indicates whether the message carries the non-roll delete attribute. Values can be:

NO

Not a non-roll delete message.

YES

Message is non-roll delete and can be deleted only by a delete operator message (DOM) instruction.

OPER

If the message is non-roll delete but is deleted only by the cursor delete function from an OCS window.

&ZMNRDRET

Indicates whether the message has been received as a result of a NRDRET command being issued by the user. This allows an NCL procedure to detect redisplayed messages and ignore them for event analysis purposes. Value is YES or NO.

&ZMOSRCID (stem.PREFIX.ORIGMSGID.ID)

Contains the message prefix of the originator of the message just received.

&ZMOSRCTP (stem.PREFIX.ORIGMSGID.TYPE)

Indicates the type of the originator of the message just received. Values can be:

null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

&ZMPPODTA

Indicates whether any PPO message profile information is available concerning this message. Value is YES or NO. If YES, then other message profile variables are available containing information relating to certain PPO attributes of the message.

&ZMPPOMSG

Indicates whether the message originated from PPO. Value is YES or NO.

&ZMPPOSEV (stem.MSGATTR.SEVERITY)

If &ZMPPODTA = YES, then the severity level of the PPO message is available in this variable. Value can be U (undeliverable), I (information), W (Important!), N (normal), or S (severe).

&ZMPPOTM (stem.SOURCE.TIME.HHMMSSTTT)

If &ZMPPODTA = YES, this variable contains the time that the message originated. Time is in the format HH.MM.SS.TTT.

&ZMPPOVNO (stem.PPOCNTL.VTAMNUM)

If &ZMPPODTA = YES, this variable contains the VTAM message number of the PPO message.

&ZMPREFXD

Indicates whether the message text has been prefixed with identifier values, for example MAI OC session identifier or ROF message prefix. Values are YES or NO.

&ZMPTEXT

Is set to the entire message text, prefixed with any ROF or MAI OC session identifiers according to the current profile settings, as it will appear on the OCS window if the message is allowed to flow to the window unchanged.

&ZMREQID

If &ZINTYPE=REQ (&INTREAD satisfied by a request message), this variable is set to the user ID of the user that issued the INTQUE command that generated the message, or to the NCL ID of the NCL process that issued the INTQUE command or the &WRITE statement. This variable is dependent on &ZINTYPE for relevance, and its setting is categorized by the &ZMREQSRC variable which will also be set. (This is available to &INTREAD only.)

&ZMREQSRC

If &ZINTYPE=REQ (&INTREAD satisfied by a request message), this variable indicates whether the process that generated the message was a user, another NCL process or a system notification. Values are USER, NCL or SYSTEM respectively. (This is available to &INTREAD only.)

&ZMSOLIC

Indicates whether the message was solicited or unsolicited. A solicited message is usually a command response. Values are YES (solicited) or NO.

&ZMSOURCE

Indicates the verb that last set the values of the message profile variables. The suite of message profile variables remain set until changed by the execution of another verb that modifies the suite. Values are:

- INTREAD
- LOGREAD
- MSGREAD

&ZMTEXT (stem.TEXT)

- Contains the text of the message received. Values are:
- Message text if the message is a standard text message
- Request message (that is, if &ZMTYPE=MSG or REQ)
- Null if the message is a delete operator message instruction (DOM)

After &LOGREAD, this variable does not include the standard log message heading information of user ID, time, and terminal name. These values are available from other message profile variables that are set after &LOGREAD.

&ZMTYPE

Specifies the type of message received after execution of the read verb. Values are:

MSG

The message is a standard text message.

DOM

The message is a delete operator message instruction.

REQ

The message is a request message that has satisfied &INTREAD TYPE=ANY or TYPE=REQ.

&INTREAD Message Profile

The message profile variables set following &INTREAD are as follows. Depending on the setting of certain key variables, some profile variables can be null.

&ZINTYPE

Specifies whether an &INTREAD operation has been satisfied by a request message or a response message. Values are REQ (request message) or RESP (response message) or NONE (no message).

&ZMALARM

Indicates whether the message will cause the terminal alarm to sound. Value is YES or NO, or null if &ZINTYPE=REQ.

&ZMALLMSG

Indicates whether the message was generated by a MSG ALL command. Value is YES or NO, or null if &ZINTYPE=REQ.

&ZMCOLOUR

Indicates the color attribute of the message. Value can be any one of NONE, RED, BLUE, GREEN, YELLOW, TURQUOISE, PINK, or WHITE. This attribute is null if &ZINTYPE=REQ.

&ZMDOM

Indicates whether the message is a delete operator message instruction. Value is YES or NO. This value is dependent also on the setting of the &ZMTYPE variable.

&ZMDOMID

Contains the delete operator message identifier (domid) of the message read, if the message has the non-roll delete message attribute, as determined by the setting of the &ZMNRD variable. If &ZMNRD=NO or &ZINTYPE=REQ, this variable is set to null.

&ZMEVONID

Is set when the incoming message was generated by an &EVENT verb and contains the nclid of the procedure which issued the &EVENT.

&ZMEVPROF

Is set for incoming event messages (N00102) and represents the EDS profile name which resulted in delivery of the event notification.

&ZMEVRCDE

Contains the routecde of the incoming event message (N00102), if the ROUTECDE operand was specified on the originating &EVENT verb.

&ZMEVTIME

Is set for incoming event messages (N00102) to the time the event originated. Time is in the format HH.MM.SS.TTT.

&ZMEVUSER

Is set when the incoming message was generated by an &EVENT verb and contains the userid of the user who issued the &EVENT verb. This variable is also set for some system events and represents the user who was responsible for the event generation.

&ZMFTSMSG

Indicates whether the message originated from the File Transmission Services (FTS) optional feature. Value is YES or NO, or null if &ZINTYPE=REQ.

&ZMHLIGHT

Indicates the display highlighting attributes of the message. Values are NONE, USCORE, REVERSE, or BLINK. This attribute is null if &ZINTYPE=REQ.

&ZMINTENS

Indicates the display intensity attribute of the message. Values are HIGH or LOW or null if no message being processed. This attribute is null if &ZINTYPE=REQ.

&ZMLNODE

Always null.

&ZMLOGCMD

Always null.

&ZMLSRCID

Contains the message prefix of the last handler of the message just received.

&ZMLSRCTP

Indicates the type of the last handler of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

This attribute is null if &ZINTYPE=REQ.

&ZMLTIME

Always null.

&ZMLUSER

Always null.

&ZMAPNAME

Contains the mapname of the object in $MSG.USERMDO for the current message (if present).

&ZMMONMSG

Indicates whether the message received is a monitor class message. Values are YES or NO. This attribute is null if &ZINTYPE=REQ.

&ZMMSG

Indicates whether the message received is a standard message or not. The setting of this variable is always opposite to the setting of the &ZMDOM variable and is dependent on the setting of the &ZMTYPE variable. This attribute is null if &ZINTYPE=REQ.

&ZMNRD

Indicates whether the message carries the non-roll delete attribute. Values can be:

NO

Not a non-roll delete message.

YES

Message is non-roll delete and can be deleted only by a delete operator message (DOM) instruction.

OPER

If the message is non-roll delete but is deleted only by the cursor delete function from an OCS window.

&ZMOSRCID

Contains the message prefix of the originator of the message just received. Will be null if &ZINTYPE=REQ.

&ZMOSRCTP

Indicates the type of the originator of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

This attribute is null if &ZINTYPE=REQ.

&ZMPREFXD

Indicates whether the message text has been prefixed with identifier values, for example MAI OC session identifier or ROF message prefix. Values are YES or NO. This attribute is null if &ZINTYPE=REQ.

&ZMPTEXT

Is set to the entire message text, prefixed with any ROF or MAI OC session identifiers according to the current profile settings, as it will appear on the OCS window if the message is allowed to flow to the window unchanged. Will be null if &ZINTYPE=REQ.

&ZMREQID

If &ZINTYPE=REQ (&INTREAD satisfied by a request message), this variable is set to the user ID of the user that issued the INTQUE command that generated the request message, or to the NCL ID of the NCL process that issued the INTQUE command. This variable is dependent on &ZINTYPE for relevance, and its setting is categorized by the &ZMREQSRC variable which will also be set. (This is available to &INTREAD only.)

This attribute is null if &ZINTYPE=RESP.

&ZMREQSRC

If &ZINTYPE=REQ (&INTREAD satisfied by a request message), this variable indicates whether the source of the INTQUE command that generated the message was a user or another NCL process. Values are USER or NCL respectively. (This is available to &INTREAD only.)

This attribute is null if &ZINTYPE=RESP.

&ZMSOLIC

Indicates whether the message was solicited or unsolicited. A solicited message is usually a command response. Values are YES (solicited) or NO. This attribute is null if &ZINTYPE=REQ.

&ZMSOURCE

Indicates the verb that last set the values of the message profile variables. It will always be INTREAD for the &INTREAD message profile. The suite of message profile variables remain set until changed by the execution of another verb that modifies the suite.

&ZMTEXT

Contains the text of the message received. This attribute is null if &ZMTYPE=DOM.

&ZMTYPE

Specifies the type of message received after execution of the read verb. Values are:

MSG

The message is a standard text message.

DOM

The message is a delete operator message instruction.

REQ

The message is a request message that has satisfied &INTREAD TYPE=ANY or TYPE=REQ.

&LOGREAD Message Profile

The message profile variables set following &LOGREAD are as follows. Depending on the setting of certain key variables, some profile variables can be null.

&ZMALARM

Indicates whether the message will cause the terminal alarm to sound. Value is YES or NO.

&ZMALLMSG

Indicates whether the message was generated by a MSG ALL command. Value is YES or NO.

&ZMCOLOUR

Indicates the color attribute of the message. Value can be any one of NONE, RED, BLUE, GREEN, YELLOW, TURQUOISE, PINK, or WHITE.

&ZMDOM

Value is always NO.

&ZMDOMID

Contains the delete operator message identifier (domid) of the message read, if the message has the non-roll delete message attribute, as determined by the setting of the &ZMNRD variable. If &ZMNRD=NO, this variable is set to null.

&ZMFTSMSG

Indicates whether the message originated from the File Transmission Services (FTS) optional feature. Value is YES or NO.

&ZMHLIGHT

Indicates the display highlighting attributes of the message. Values are NONE, USCORE, REVERSE, or BLINK.

&ZMINTENS

Indicates the display intensity attribute of the message. Values are HIGH or LOW or null if no message being processed.

&ZMLNODE

Indicates the terminal name of the user to whom a log message is to be attributed. Value is the name of a terminal. (This is available to &LOGREAD only.)

&ZMLOGCMD

Indicates whether a log message is an echo to the log of a command. Value is YES or NO. (This is available to &LOGREAD only.)

&ZMLSRCID

Contains the message prefix of the last handler of the message just received.

&ZMLSRCTP

Indicates the type of the last handler of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

&ZMLTIME

Contains the timestamp of a log message. Time is in the format HH.MM.SS.TTT.

&ZMLUSER

Contains the user ID to whom generation of the log message is to be attributed.

&ZMAPNAME

Contains the mapname of the object in $MSG.USERMDO for the current message (if present).

&ZMMONMSG

Indicates whether the message received is a monitor class message. Values are YES or NO.

&ZMMSG

Value is always YES.

&ZMNRD

Indicates whether the message carries the non-roll delete attribute. Values can be:

NO

Not a non-roll delete message.

YES

Message is non-roll delete and can be deleted only by a delete operator message (DOM) instruction.

OPER

If the message is non-roll delete but is deleted only by the cursor delete function from an OCS window.

&ZMOSRCID

Contains the message prefix of the originator of the message just received.

&ZMOSRCTP

V the type of the originator of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

&ZMPREFXD

Indicates whether the message text has been prefixed with identifier values, for example MAI OC session identifier or ROF message prefix. Values are YES or NO.

&ZMPTEXT

Value is always null.

&ZMREQID

Value is always null.

&ZMREQSRC

Value is always null.

&ZMSOLIC

Indicates whether the message was solicited or unsolicited. A solicited message is usually a command response. Values are YES (solicited) or NO.

&ZMSOURCE

Indicates the verb that last set the values of the message profile variables. This will always be LOGREAD for the &LOGREAD message profile. The suite of message profile variables remain set until changed by the execution of another verb that modifies the suite.

&ZMTEXT

Contains the text of the message received.

After &LOGREAD, this variable does not include the standard log message heading information of user ID, time, and terminal name. These values are available from other message profile variables that are set after &LOGREAD.

&ZMTYPE

Value is always MSG.

&MSGREAD Message Profile

The message profile variables set following &MSGREAD are as follows. Depending on the setting of certain key variables, some profile variables can be null.

&ZMALARM

Indicates terminal alarm. Values are YES or NO.

&ZMALLMSG

Indicates whether the message was generated by a MSG ALL command. Value is YES or NO, or null if &ZMTYPE=DOM.

&ZMCOLOUR

Indicates the color attribute of the message. Value can be any one of NONE, RED, BLUE, GREEN, YELLOW, TURQUOISE, PINK, or WHITE. This attribute is null if &ZMTYPE=DOM.

&ZMDOM

If &ZMTYPE=DOM, value is DOM; otherwise, it is NO.

&ZMDOMID

Contains the delete operator message identifier (domid) of the message read, if the message has the non-roll delete message attribute, as determined by the setting of the &ZMNRD variable. If &ZMNRD=NO, this variable is set to null.

&ZMFTSMSG

Indicates whether the message originated from the File Transmission Services (FTS) optional feature. Value is YES or NO. This attribute is null if &ZMTYPE=DOM.

&ZMHLIGHT

Indicates the display highlighting attributes of the message. Values are NONE, USCORE, REVERSE, or BLINK. This attribute is null if &ZMTYPE=DOM.

&ZMINTENS

Indicates the display intensity attribute of the message. Values are HIGH or LOW or null if no message being processed. This attribute is null if &ZMTYPE=DOM.

&ZMLOGCMD

Value is always null.

&ZMLNODE

Value is always null.

&ZMLSRCID

Contains the message prefix of the last handler of the message just received. Values can be:

Null

Message generated internally by this system.

ROF msgid

Message prefix of the ROF session which delivered the message.

MAIOC sessionid

The session ID of the MAI OC session that delivered the message.

This attribute is null if &ZMTYPE=DOM.

&ZMLSRCTP

Indicates the type of the last handler of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

This attribute is null if &ZMTYPE=DOM.

&ZMLTIME

Value is always null.

&ZMLUSER

Value is always null.

&ZMAPNAME

Contains the mapname of the object in $MSG.USERMDO for the current message (if present).

&ZMMONMSG

Indicates whether the message received is a monitor class message. Values are YES or NO.

&ZMMSG

Value is YES or NO.

&ZMNRD

Indicates whether the message carries the non-roll delete attribute. Values can be:

NO

Not a non-roll delete message.

YES

Message is non-roll delete and can be deleted only by a delete operator message (DOM) instruction.

OPER

If the message is non-roll delete but is deleted only by the cursor delete function from an OCS window.

&ZMOSRCID

Contains the message prefix of the originator of the message just received. Will be null if &ZMTYPE=DOM.

&ZMOSRCTP

Indicates the type of the originator of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

This attribute is null if &ZMTYPE=DOM.

&ZMPREFXD

Indicates whether the message text has been prefixed with identifier values, for example MAI OC session identifier or ROF message prefix. Values are YES or NO. This attribute is null if &ZMTYPE=DOM.

&ZMPTEXT

Is set to the entire message text, prefixed with any ROF or MAI OC session identifiers according to the current profile settings, as it will appear on the OCS window if the message is allowed to flow to the window unchanged. This attribute is null if &ZMTYPE=DOM.

&ZMREQID

Value is always null.

&ZMREQSRC

Value is always null.

&ZMSOLIC

Indicates whether the message was solicited or unsolicited. A solicited message is usually a command response. Values are YES (solicited) or NO. This attribute is null if &ZMTYPE=DOM.

&ZMSOURCE

Indicates the verb that last set the values of the message profile variables. This will always be MSGREAD for the &MSGREAD message profile. The suite of message profile variables remain set until changed by the execution of another verb that modifies the suite.

&ZMTEXT

If &ZMTYPE=MSG, &ZMTEXT contains the entire message text. If &ZMTYPE=DOM, then &ZMTEXT is null.

&ZMTYPE

Specifies the type of message received after execution of the read verb. Values are:

MSG

The message is a standard text message.

DOM

The message is a delete operator message instruction.

REQ

The message is a request message that has satisfied &INTREAD TYPE=ANY or TYPE=REQ.

The value DOM is possible only if DOM=YES is coded on the &MSGREAD statement.

&PPOREAD Message Profile

The message profile variables set following &PPOREAD are private to the PPOPROC system procedure, and are different from the variables that form the message profile for the &INTREAD, &LOGREAD and &MSGREAD verbs.

&PPOALERT

Value is YES if this message was routed to PPOPROC as a result of a &PPOALERT verb. &PPOALERT can be used to send messages to PPOPROC either locally or in remote systems. If the message was not originated by a &PPOALERT then value is NO.

&PPOCOLOR

Value is set according to the SYSPARMS PPOCOLOR= operand.

&PPODEFM

Value is YES if the message number has been defined by the DEFMSG command for delivery to PPOPROC. This variable can be set to NO if delivery for the UNSOLICITED class of messages PPOPROC delivery IS set but specific delivery for this message number is NOT.

&PPODOMID

The delete operator message identifier (DOMID) of the message if the PPO message is non-roll delete. Value is DOMID if &PPONRD = YES; otherwise, this variable is set to null value.

&PPODLOC

Value is YES if the message number is one that has been defined through the DEFMSG command for LOCAL delivery. Value is NO if message has not been defined for delivery to local receivers.

&PPODREM

Value is YES if the message number is one that has been defined through the DEFMSG command for REMOTE delivery. Value is NO if message has not been defined for delivery to remote receivers.

&PPOFIRST

Indicates the first message from a possible multi-line message block delivered to PPOPROC. The purpose of this variable is to indicate a new VTAM message group but not necessarily the first message of the group delivered by VTAM as these might not be eligible for delivery to PPOPROC.

&PPOHLITE

Value is set according to the SYSPARMS PPOHLITE= operand.

&PPOLDID

Domain ID of the last region to handle this PPO message. If generated by the local system, this variable will contain the domain ID of the local system. The value can be different from the originating domain ID if the message was onward routed by an intermediate system.

&PPOMSGNO

The VTAM message number of the PPO message just received. Note that this value is also available in the &ZMPPOVNO message profile variable following a &INTREAD, &LOGREAD, or &MSGREAD operation.

&PPOMSGSV

The VTAM message severity level associated with the PPO message just received. Value can be U (undeliverable), I (information), W (Important!), N (normal), or S (severe).

Note that the value is also available in the &ZMPPOSEV message profile variable following a &INTREAD, &LOGREAD, or &MSGREAD operation.

&PPONRD

Value is YES if the PPO message has the non-roll delete attribute (which occurs if a reply is required to the message). Otherwise, value is NO.

&PPOODID

The domain ID of the region from which the PPO message originated.

&PPOONETN

The network name of the VTAM that generated the PPO message.

&PPOONMID

The region ID (from the SYSPARMS ID= operand) of the system that generated the message.

&PPOPRIRN

If the PPO message contains qualified network resource names this is the primary NETWORK name.

Your product analyzes message text based on VTAM tables to extract resource names. This extraction does not necessarily reflect SNA class resources, and can vary with the version of VTAM.

CA recommends that resource extraction be performed at message level by PPOPROC, as the extracted values might not reflect the true resource hierarchy in the VTAM multi-line messages.

&PPOPRIRS

If the PPO message contains network resource names this is the primary resource name. In the context of PPOPROC, this is the resource name of the first message in a VTAM multi-line display.

Your product analyzes message text based on VTAM tables to extract resource names. This extraction does not necessarily reflect SNA class resources, and can vary with the version of VTAM.

CA recommends that resource extraction be performed at message level by PPOPROC, as the extracted values might not reflect the true resource hierarchy in the VTAM multi-line messages.

&PPOSECRS

If the PPO message contains network resource names this is the secondary resource name. If the message is part of a VTAM multi-line message, this is the first resource name in the current message.

&PPOSECRN

If the PPO message contains qualified network resource names, this is the secondary NETWORK name.

&PPOOSSCP

The SSCP name of the VTAM that generated the PPO message.

&PPOTEXT

The PPO message text.

&PPOTIME

The PPO message generation time in HH.MM.SS.TTT format. For messages delivered across an ISR link it reflects time from the remote system converted to the time on the local system.

&PPOTYPE

Defines the type of PPO message received. Value is PPO if the message is a standard unsolicited message delivered from VTAM, SPO if the message is actually the reply to a command issued by an OCS operator or NCL procedure or CMD if the message is a copy of a command entered by an OCS operator or NCL procedure.

&PPOUID

The user ID associated with a message or command which has &PPOTYPE = CMD or &PPOTYPE = SPO, or for which &PPOALRT = YES.

&PPOVMSG

Indicates (YES or NO) whether a message was found as a VTAM message in the DEFMSG table.

Appendix C: Sample APPC Conversations

This section contains the following topics:

Sample Conversations Between Two Systems

The purpose of the sample APPC conversations is to illustrate the use of the &APPC NCL verb. The samples are provided in the distribution library. A complete list of sample conversations and detailed instructions on how to run them are also given in the member $SAAPDOC.

Source and Target NCL Procedures

When two NCL procedures communicate using an APPC connection, the one requesting the connection (via the &APPC ALLOCATE verb) is called the source procedure. As a result of the allocation request, a target procedure is started or attached in the remote system. All the sample conversations involve a source and a target procedure pair.

To allocate a conversation, the source procedure issues an &APPC ALLOCATE verb specifying the appropriate transaction identifier. In the sample conversations, source and target NCL procedure pairs are denoted by $SAAPSxx and $SAAPTxx respectively and the associated transaction identifier by $SATRNxx.

Run the Sample APPC Conversations

To run the sample APPC conversations, choose one or more of the transactions available ($SATRNxx) and one of the following environments. The sample conversations are assumed to run between a source system (NMA) and a target (NMB). If you want to run the samples, you should replace these by the LU names of your particular systems.

Running the APPC samples involves the following steps:

- APPC environment definitions. This involves defining transactions and APPC links to both the source and the target systems. Examples of three different environments are provided.
- Running the APPC transactions. This involves starting the source procedure $SAAPSxx in the source system NMA. Information about the completion of &APPC verbs (from &RETCODE and &ZFDBK), the 'what-received' indicator (from &ZAPPCWR) and the state of the conversation (from &ZAPPCSTA) is reported in messages written to the system log. A description of the first three sample conversations is given.

Environment 1: Local Conversations

This environment involves only one system. Both the source procedure $SAAPSxx and the target procedure $SAAPTxx run in the same system and the same NCL region. Since no LU6.2 sessions are established no link definitions are necessary. Environments 1 and 2 are the simplest in which to run the sample conversations and require only transaction definitions, as follows:

```
DEFTRANS TRANSID=$SATRNxx PROC=$SAAPTxx
```

The default destination information for the transaction is omitted, causing the ENV=CURRENT parameter to be assumed, which indicates that $SAAPTxx is to be started in the same system and the same NCL region as the source procedure.

Environment 2: Same LU Conversations

This environment involves only one system. Both the source procedure $SAAPSxx and the target procedure $SAAPTxx run in the same system. In this case, however, the target procedure is started in the background server environment, BSVR. As no LU6.2 sessions are established, no link definitions are necessary. Environments 1 and 2 are the simplest in which to run the sample conversations and require only transaction definitions, as follows:

```
DEFTRANS TRANSID=$SATRNxxLU=NMA PROC=$SAAPTxx
```

The operand LU=NMA indicates that $SAAPTxxis to be started in the same system as the source procedure.

Environment 3: Conversations Between Two Systems

In this environment, the source procedure, $SAAPS*xx*, is started in NMA and the target procedure, $SAAPT*xx*, is started in NMB. A single session APPC link is established between NMA and NMB and both link and transaction definitions are needed.

Definitions in the source system NMA:

```
DEFTRANS TRANSID=$SATRNxx LINK=NMB
DEFLINK TYPE=APPC LINK=NMB LU=NMB MON=YES
```

Definitions in the target system NMB:

```
DEFTRANS TRANSID=$SATRNxx PROC=$SAAPTxx
DEFLINK TYPE=APPC LINK=NMA LU=NMA
```

Appendix D: NDB Response Codes

This section contains the following topics:

About Response Codes

The response codes fall into three categories:

- All OK (response 0)
- Warning conditions (1 to 29)
- Error conditions (30 to 255)

An NCL procedure terminates if an error response is returned, unless &NDBCTL ERROR=CONTINUE is in effect, except as follows:

- &NDBOPEN is always processed as if &NDBCTL ERROR=CONTINUE is in effect, except for response 34 (already open), which honors the actual setting of &NDBCTL ERROR.
- &NDBCLOSE is always processed as if &NDBCTL ERROR=CONTINUE is in effect, except for response 35 (not open), which honors the actual setting of &NDBCTL ERROR.

These exceptions allow a procedure to recover from any error conditions encountered when opening or closing an NDB, except for the obvious programming error (already open or closed). This should encourage you to develop exception handling logic when first writing a procedure.

If an error response from an &NDBOPEN is ignored, the next &NDB statement referring to that NDB will cause the procedure to abnormally terminate (as a result of trying to access an NDB that has not been opened).

Error Information

Each response code description includes an indication of the information that will be provided in the &NDBERRI system variable. N/A means there is no information provided, and &NDBERRI is null.

Response Codes

The response codes are listed in ascending order:

0

ERRI=N/A

Database processing completed with no errors.

1

ERRI=N/A

Record with requested key or RID not found (GET/UPD/DEL), OR reposition RID not in scan result list (SEQ RESET).

2

ERRI=N/A

End of file on GET sequential (forward or backward).

3

ERRI=field name

Named field not found for INFO on field name.

4

ERRI=N/A

No records found for supplied SCAN criteria.

5

ERRI=N/A

SCAN terminated-exceeded I/O limit.

6

ERRI=N/A

SCAN terminated-exceeded time limit.

7

ERRI=N/A

SCAN terminated-exceeded storage limit.

8

ERRI=N/A

SCAN terminated-met or exceeded number of records limit.

10-19

ERRI=N/A

Linked &NDBGET/&NDBFMT-extra no find return code as used in the link definition.

20

ERRI=format name

FMT del, format name not found.

21

ERRI=sequence name

SEQ del, sequence not found.

29

ERRI=N/A

Maximum possible Important! code-currently not assigned.

30

ERRI=N/A

Error in request (catch-all). See command/messages.

31

ERRI=N/A

Logic error in request: not signed on to database (NDB). For example, if the NCL procedure is paused and the database stopped/started, then the next call gets this error.

32

ERRI=N/A

Insufficient storage to process request.

33

ERRI=N/A

OPEN request rejected, database is stopping.

34

ERRI=N/A

&NDBOPEN-already open to this NDB.

35

ERRI=N/A

&NDBCLOSE-not open to this NDB.

36

ERRI=N/A

NDB PURGE-nominated NDB not found, could not purge.

37

ERRI=N/A

NDB PURGE-nominated NDB not locked, cannot purge.

38

ERRI=N/A

Load for load module for requested NDB function failed.

39

ERRI=N/A

Requested option not allowed-NDB is in load mode.

40

ERRI=N/A

&NDBOPEN failed by NCLEX01 user exit.

41

ERRI=N/A

This &NDB... verb blocked by &NDBOPEN user exit (NCLEX01) return mask.

51

ERRI=N/A

Database open failure: VFS open for database failed.

Possible causes:

- File not allocated to your product.
- UDBCTL OPEN not done or in error.

52

ERRI=N/A

Database open failure: Specified database name is not a VSAM KSDS.

53

ERRI=N/A

Database open failure: VSAM relative key position (RKP) not 0.

54

ERRI=N/A

Database open failure: VSAM key length too short (that is, fewer than 16) to be a valid NDB database.

55

ERRI=N/A

Database open failure: VSAM data length too short to be a valid NDB database.

56

ERRI=N/A

Database open failure: Read of control record failed.

57

ERRI=N/A

Database open failure: Control record not valid.

58

ERRI=N/A

Database open failure: Unable to get storage to contain control record.

59

ERRI=N/A

Database open failure: Read of transaction control record failed.

60

ERRI=N/A

Database open failure: Transaction control record not valid.

61

ERRI=N/A

Database open failure: Unable to get storage to contain transaction control record.

62

ERRI=N/A

Database open failure: Unable to get storage to contain transaction data record.

63

ERRI=N/A

Database open failure: Error building field name indexes.

64

ERRI=N/A

Database open failure: Error re-applying pending transaction. Probable file full condition.

65

ERRI=N/A

Database open failure: Database flagged as in DEFER status. Probable file corrupt condition, system has failed with database started in DEFER mode.

66

ERRI=N/A

Database open failure: UDB is open by other users.

67

ERRI=N/A

UDB is open INPUT and NDB START or &NDBOPEN is not for INPUT only mode.

68

ERRI=N/A

Domain ID on NDB control record and this system mismatch, and FORCE not on NDB START. NDB is not started. Important!-it might be currently open in another system.

69

ERRI=N/A

NDB version in this NDB control record not supported.

70

ERRI=N/A

This NDB is open under another file ID on this system (VSAM timestamp match).

71

ERRI=N/A

Request not allowed on active database. For example, NDB CREATE, NDB RESET, NDB ALTER, or NDB UNLOAD.

72

ERRI=N/A

User ID required for request. Internal failure, should never occur with NDB command.

73

ERRI=sequence name

Specified sequence name not defined.

74

ERRI=sequence name

Sequence name required or already defined.

75

ERRI=N/A

Free-format text had an invalid token:

- Too long
- Unmatched quotes

76

ERRI=N/A

Logic error-not signed on to database (NDB). For example, if NCL proc is paused, and DB stopped/started, then next call will get this error.

77

ERRI=N/A

This user ID has an asynchronous request running. Internal failure; should never occur with NDB command or &NDB statements.

78

ERRI=N/A

Request internally canceled.

79

ERRI=N/A

OPEN EXCLUSIVE-other users on database.

80

ERRI=N/A

OPEN-database locked by an exclusive user.

81

ERRI=field name if relevant

Add, update, or delete field, an error in the field list syntax, and so on.

82

ERRI=N/A

Add, update, or delete field, an internal error in the field index.

83

ERRI=field name

Value for named field is bad, not recognizable as valid data.

84

ERRI=field name

Value for named field is too long.

85

ERRI=field name

Value for named numeric field is not numeric, or outside range:

-2,147,483,648 : 2,147,483,647.

86

ERRI=field name

Value for named hexadecimal field is not a valid hexadecimal string, or is an odd number of characters.

87

ERRI=field name

Value for named DATE or CDATE field is not valid.

88

ERRI=field name

Field name is not a valid floating point number.

89

ERRI=field name

Supplied value is not a valid hexadecimal number.

90

ERRI=field name

Value for named time field is not a valid time, in the format:

HHMMSS or HHMMSS.TTTTTT.

92

ERRI=N/A

Operation not allowed. NDB or user open in input-only mode.

93

ERRI=field name

Field update request-could not read field record from NDB.

94

ERRI=field name

Supplied value is not a valid timestamp, in the format:

YYYYMMDDHHMMSS.TTTTTT.

101

ERRI=token in error

ADD or UPD, field=value list syntax error.

102

ERRI=field name

ADD or UPD, required field not provided.

103

ERRI=field name

ADD record, sequence key value already on database (that is, not unique).

104

ERRI=field name

ADD/UPD record, KEY=UNIQUE field value not unique.

105

ERRI=field name

UPD record, UPDATE=NO value change for field.

106

ERRI=format name

Format specified on an ADD or UPD does not exist.

107

ERRI=format name

Format specified on an ADD or UPD exists but is an INPUT format and cannot be used for output.

111

ERRI=format name

FMT add, format name '*' is invalid.

112

ERRI=format name

FMT add, format name already exists.

114

ERRI=token in error

FMT/GET, format list syntax error.

115

ERRI=field name

FMT/GET, field name not defined in database.

116

ERRI=format name

Format specified on a GET exists but is an OUTPUT format and cannot be used for input.

117

ERRI=field name

You have referenced an NDB field twice in an OUTPUT format definition.

118

ERRI=field name

You have used an NCL keyword name in an OUTPUT format, but the keyword is not an NCL system variable name.

121

ERRI=format name

GET, supplied format name not defined.

122

ERRI=field name

GET by key, key field not defined on database.

123

ERRI=field name

GET by key, field not keyed.

124

ERRI=sequence name

GET by sequence, sequence ID not defined.

125

ERRI=sequence name

GET by sequence, key field for sequence has been deleted (by &NDBDEF DELETE).

126

ERRI=sequence name

GET by sequence, skip=0 specified and not currently positioned.

127

ERRI=sequence name

GET by sequence, skip=0 specified and currently at EOF (front or back).

128

ERRI=field name

GET by key field, GENERIC= is invalid for this field format. Generic access is only allowed for character and HEX fields.

130

ERRI=field name

&NDBGET histogram (KEY=) on sequence key not supported.

131

ERRI=sequence name

SEQ DEF-invalid sequence name.

132

ERRI=sequence name

SEQ DEF, sequence name already exists.

133

ERRI=N/A

SEQ DEF, from or to RID invalid.

134

ERRI=sequence name

SEQ DEF, field name not found or is not a key.

135

ERRI=N/A

SEQ DEF, from or to values not valid.

136

ERRI=field name

SEQ DEF, generic invalid with field format.

137

ERRI=N/A

SEQ DEF, invalid null value for GENERIC=.

139

ERRI=sequence name

SEQ RESET, sequence not found.

140

ERRI=invalid data value

SEQ RESET, REPOS= value invalid or null.

141

ERRI=N/A

SEQ RESET, REPOS= not valid on a scan sequence that is not sorted, or, while sorted, the primary sort field was substringed.

142

ERRI=sequence name

SEQ RESET, REPOS by RID only valid for a SCAN sequence.

143

ERRI=N/A

&NDBSEQ histogram (KEY=) on sequence key not supported.

144

ERRI=sequence name

RELPOS sequence is not a sequence constructed by &NDBSCAN.

151

ERRI=N/A

INFO by field number, number lt 1 or gt number fields in database.

161

ERRI=sequence name

SCAN sequence ID is already defined.

162

ERRI=sequence name

SCAN sequence ID is currently in use, by an active scan for this user.

163

ERRI=field name

SCAN SORT= field name not defined on database.

183

ERRI=N/A

SCAN syntax error in scan request.

191.

ERRI=N/A

Unload failed. Accompanying messages will indicate the cause of the failure

193

ERRI=N/A

NDB ALTER failed.

237

ERRI=N/A

File integrity error, get if continuation DBDR failed. Database possibly corrupted. Contact Technical Support.

238

ERRI=field name

File integrity error, get XFF if DBKR failed. Database possibly corrupted. Contact Technical Support.

239

ERRI=field name

File integrity error, get KGE if DBKR failed. Database possibly corrupted. Contact Technical Support.

240

ERRI=N/A

VSAM I/O Error. See log for more information.

241

ERRI=N/A

Request not processed-invalid request code (internal error), should never occur with NCL (&NDB) or NDB command.

242

ERRI=N/A

Request not processed-RPL busy flag set (internal error), should never occur with NCL (&NDB) or NDB command.

243

ERRI=N/A

Request not processed-insufficient storage to queue request to database handler. Try again.

244

ERRI=N/A

Request not processed-required text parameter not provided (internal error). Should never occur with NCL (&NDB) or NDB command.

245

ERRI=N/A

Request not processed-Database not started.

246

ERRI=N/A

Request not processed-required VAL1 parameter not provided (internal error). Should never occur with NCL (&NDB) or NDB command.

247

ERRI=N/A

Request not processed-required VAL2 parameter not provided (internal error). Should never occur with NCL (&NDB) or NDB command.

248

ERRI=N/A

Request not processed-database not LOCKED. Only applicable to NDB START UNLOCK command.

250

ERRI=N/A

Request not processed-database is LOCKED or STOPPING.

251

ERRI=N/A

Request not processed-long running command currently in progress. For example, NDB UNLOAD.

252

ERRI=abend code

This return code indicates that an NDB operation was terminated due to a logical abend caused by a possible NDB corruption. The log will contain useful debugging information.

For update requests, the NDB will be stopped and locked. For read/scan requests, the request is terminated but the NDB will continue to process other requests.

254

ERRI=N/A

Request not processed-feature not present, or your product is shutting down.

Appendix E: Using Key Ranges with an NDB

If an NDB contains large amounts of data, or if you want to separate the data records from the key and control records, you can use the KEYRANGES AMS parameter. The information about the key structure of an NDB enables you to do separate the records.

Note: This information is for guidance only. The key structures can alter with future product releases. If you divide NDBs by key ranges, review this information whenever you install a new release of your product.

This section contains the following topics:

NDB Key Structure

The VSAM key in an NDB is divided into the following parts:

- A type prefix, two bytes long
- For some records, a field code, two bytes long
- Key data, either the rest of the key, or four bytes shorter
- For some keys, a suffix, four bytes long

The following shows the key structures in ascending order:

- Control record (always first record in data set)

  Rectype: X'0000'

  Rest of key: All binary 0
- Transaction control record (always second record in data set)

  Rectype: X'0010'

  Rest of key: All binary 0
- Transaction journal records

  Rectype: X'0011'

  Sequence number: 2 bytes, binary (X'0001' to nnnn)

  Rest of key: All binary 0

- Field information records (2 types)
  - First (Basic Field Information):

    Rectype: X'0020'

    Field code: 2 bytes binary

    Rest of key: All binary 0
  - Second (Contains Key Statistics):

    Rectype: X'0021'

    Field code: 2 bytes binary

    Rest of key: All binary 0
- Key records

  Rectype: X'0030'

  Field code: 2 bytes binary

  Field value: (keylen - 8 bytes, see Storage of Values)

  Suffix: 4 bytes, binary
- RID to sequence key records (present only if NDB defined with a sequence key)

  Rectype: X'0032'

  RID: 4 bytes, binary

  Rest of key: All binary 0
- Data records. There are 2 formats, depending on whether the NDB has a sequence key, or not:
  - For databases defined without a sequence key, the format is:

    Rectype: x'0040'

    RID: 4 bytes, binary

    Reserved: (keylen - 10) bytes long

    Sequence number: 4 bytes binary (usually 0, used to handle large data records)
  - For databases defined with a sequence key, the format is:

    Rectype: x'0040'

    field code: 2 bytes, binary; always x'0001'

    Seq key value: (keylen - 8 bytes, see Storage of Values)

    Sequence number: 4 bytes binary (usually 0, used to handle large data records)

Storage of Values

Field values and sequence key values are stored as follows:

CHAR fields

The character field value, padded to (keylen-8) with blanks.

NUM fields

4 bytes, binary, with the sign bit inverted (that is, 0 is stored as X'80000000', 100 is X'80000064'). Padded to (keylen-8) with binary 0.

HEX fields

Stored in compressed hexadecimal format. Padded to (keylen-9) with binary 0. The last byte (of the value part, not of the total key) contains the significant length in binary (so X'ABCD' and X'ABCD00' are distinct; X'ABCD' is stored as X'ABCD0000...02' and X'ABCD00' is stored as X'ABCD0000...03').

DATE fields

Stored in unsigned packed. For example, 21 September 2004 is stored as X'040921'. Padded to (keylen-8) with binary 0.

FLOAT fields

Stored in 8-byte floating point, with the following change to force character compares to work correctly:

- If sign bit is 0 (that is, number is 0 or positive), invert the sign bit.
- If the sign bit is 1 (that is, a negative number), invert the entire 8-byte field.
- Key length padding is binary 0.

CDATE fields

Stored internally in 3-byte binary with value 1 (x'000001') representing 1/1/0001. Key length padding is binary 0.

TIME fields

Stored internally in 5-byte binary as a number of micro-seconds, values from 0 to 86,399,999,999 (86,400 seconds in a day, times 1000000 for microseconds, -1 microsecond). The largest hexadecimal value is x'141DD75FFF'. Key length padding is binary 0.

TIMESTAMP fields

Stored internally as a concatenation of the CDATE (3-byte) and TIME (5-byte) fields (CDATE first). Thus a TIMESTAMP takes 8-bytes. Key length padding is binary 0.

Suggested Key Ranges

The following are suggested key range breakups:

X'0000' to X'002F'

Includes the control, transaction, and field records. The transaction records have the most activity. The field records can be in the same key range, as they are only referenced on NDB START, or when an &NDBDEF statement is processed.

X'0030' to X'003F'

Includes all the key records, and, for databases with a sequence key, the RID to sequence key relation records.

Splitting the key records by field is not feasible because the field code is not readily determined. If fields have been added or deleted, the code can change if an NDB is unloaded and reloaded. The only field code that is guaranteed is the field code for a sequence key, which is always X'0001'.

If field codes *are* necessary for splitting key records, the field information records (record type X'0020') contain the field code in bytes 3 through 4 of the key. The code is also in the data, in the 2 bytes immediately following the key. The next 12 bytes after the field code in the data contain the field name. Remember that this field code can be different if an NDB is unloaded and reloaded.

X'0040' to X'FFFF'

Contains all data records.

For NDBs with a sequence key, the data can be broken on sequence key value by preceding the value limits with X'00400001'. For example, if an NDB has a character sequence key, and you want to break A-K, and L-Z, you can use:

```
X'00400001C1' to X'00400001D2' (A-K)
X'00400001D3' to X'00400001E9' (L-Z)
```

Other Considerations

Failure to provide complete key ranges could lead to VSAM errors if a key value cannot be matched to a key range.

Using key ranges other than to separate control, key, and data records, or, for NDBs with a sequence key, sequence key ranges, is not recommended.

Appendix F: Using NCLEX01 for NDB Security

A user-nominated NCLEX01 exit can be invoked if SYSPARMS NDBOPENX is set to YES. In this case, all &NDBOPEN statements that actually open the NDB (for this process) result in the nominated NCLEX01 exit being called.

This section contains the following topics:

NDB Open Exit Call Details

The following describes the parameter list passed on an &NDBOPEN call to NCLEX01:

- The standard parameter list, as mapped by the $NMNCEX1 macro.
- This parameter list contains information common to all calls to NCLEX01 for various reasons. It contains a function code, and information about the current user ID, and so on. The security correlator is passed to allow access to any security information provided by any security exits.
- On an &NDBOPEN call, the NEXFUNC field has a value of 12 (decimal).
- Field NEXNDEX1 contains a pointer to a supplementary parameter list containing additional information.
- A supplementary parameter list, mapped by the NEXND DSECT. This DSECT is in the NCLEX01 macro.

Fields in this DSECT include:

NEXNDNAM	DS CL8		NDB name being opened
NEXNDFL1	DS X		Flag byte
NEXNDOPX	EQU	X'80'	... EXCLUSIVE on &NDBOPEN
NEXNDOPI	EQU	X'40'	... INPUT on &NDBOPEN
NEXNDOKM	DS	0XL2	Following 2 bytes. Note: initialized to 2X'FF'.
NEXNDOK1	DS	X	... first OK operations flag.
NEXNDGTO	EQU	X'80'	... 1 = &NDBGET ok.
NEXNDSCO	EQU	X'40'	... 1 = &NDBSCAN ok.

NEXNDADO	EQU	X'20'	... 1 = &NDBADD ok.
NEXNDUPO	EQU	X'10'	... 1 = &NDBUPD ok.
NEXNDDLO	EQU	X'08'	... 1 = &NDBDEL ok.
NEXNDDFO	EQU	X'04'	... 1 = &NDBDEF ok.
NEXNDINO	EQU	X'02'	... 1 = &NDBINFO ok.
NEXNDOK2	DS	X	... 2nd ok operations flag.
NEXNDUDL	DS	H	Length (0-50 decimal) of user data from &NDBOPEN statement.
NEXNDUDT	DS	CL50	User data (padded/truncated to 50 chars) from &NDBOPEN statement (pad is blank).

These fields can be referenced to determine the action to take.

By returning R15 not equal to 0, all access to the NDB from the NCL process is denied. ASN NDB response code of 40 is returned, and this is handled based on the &NDBCTL ERROR= setting.

If R15 is returned with a 0, the 2-byte NEXNDOKM field is examined. If all of the defined bits are off (as listed previously), then the effect is the same as setting R15 not 0. Note that the NEXNDOKM field is initialized to all bits on before the call.

If any bits in the NEXNDOKM field are off, the associated &NDBxxxx verb cannot be used. Attempts to use it results in an NDB response code of 41. This too is handled as per the &NDBCTL ERROR= setting.

You can deny the use of the &NDBDEF verb with this exit. This forces the use of the NDB FIELD command to add, update, or delete field definitions from the database.

If the exit terminates abnormally while processing the &NDBOPEN request, the NCL process is terminated with an error.

The exit is attached as a subtask in all environments. It should be coded re-entrant because multiple copies may be executing simultaneously.

Appendix G: Using the Batch Command Interface

This section contains the following topics:

Batch Command Interface

You invoke the Batch Command Interface (BCI) by executing the NMBCI program in a batch processing environment or by calling it from other programs.

Note: The BCI is available on z/OS systems only.

Commands are read from a nominated input device (in card image format) and the replies received are routed to the nominated output device. A user command exit can also be called where commands may be added, changed, or deleted. Return codes from an executed NCL procedure may be translated to JCL return codes, which can then be interpreted using conditional JCL statements.

Requests are executed by NCL procedures running under the User Services option, and the unformatted results of these requests are returned to the output device.

This interface may be used to connect to a product region operating within the same or within a different VTAM domain.

BCI Command Types

Commands read from an input file fall into two categories:

- BCI control commands used to provide details for session establishment, to request activation of a BCI command exit, and to disconnect the session. These commands are described in a following section.
- Any other commands, which are routed unchanged to the target region for interpretation and execution. These are processed by NCL procedure $USERBCI, which runs under the User Services option.

Once a BCI command exit has been activated, all non-BCI input commands are passed for pre-processing before continuing. The exit can add new control or user commands, and can modify command contents or delete commands.

BCI Input

Commands are extracted from an input file in card-image format. The input file is referenced by the SYSIN DD statement and may be any LRECL=80 data set.

Only the first 72 columns of the record can contain command data; the remainder of the record is ignored. A command may be preceded by one or more spaces and is terminated by a space. Any data (including additional blanks) after the command, is assumed to be operands or parameters associated with the command.

BCI Output

An activity log is directed to the nominated output file and consists of 121-character print lines where the first byte is a machine control character. The file is referenced by the SYSOUT DD statement and should be specified as RECFM=FBM with an appropriate block size.

The activity log records the commands processed by the BCI, the user commands (after insertion or modification by any command exit), the replies from the User Services procedure, and any messages issued by the BCI itself.

The column headed Terminal indicates the device that issued the command as one from the following:

- The system input file name
- The LU name of the virtual terminal used for the session
- The name of the command exit (if the command is added or modified)

An additional flag character preceding the command or message reply indicates the source as follows:

+ Indicating it was read from the system input file.

- Shows it was inserted or changed by the command exit.

= Indicates a message issued by the BCI.

A blank indicates a reply from the User Services procedure.

BCILOGON Command—Establish Session

The BCILOGON command establishes a session with the desired product region on behalf of the specified user.

This command has the following format:

```
BCILOGON applname
         USER userid ( /password | PASSWORD password )
       [ MENU menu ]
       [ LUNAME applid | LUPREF luprefix ]
```

applname

Is the VTAM APPL name of the product region to which you want to connect. This value must be the first parameter following the BCILOGON command.

USER *userid* { /*password* | PASSWORD *password* }

Provides the valid user ID and password combination to use for the session. The user ID and password can be specified in a single string after the USER keyword (that is, as USER *userid*/*password*) or provided separately (that is, as USER *userid* PASSWORD *password*). Both the user ID and password must be specified if BCILOGON is run as a batch job on demand.

If accessing BCI from another program, the user ID and password for that system (as taken from the UAMS database or the equivalent security exit procedure) are defaulted for the interface. Logging on to BCI is transparent, without you having to provide user ID and password, or the ID of the user who submitted the batch job.

MENU *menu*

Specifies the initial menu selection data for the User Services facility. This value can contain up to five characters. If present, the characters U.menu are passed to the product region to perform initial menu selection.

If omitted, the User Services procedure defined for the BCI user ID is selected (this should be $USERBCI in the BCI facility distributed). However, if the user is set up to use the standard $USERSER procedure, a menu option of BCI can be specified.

LUNAME *applid* or LUPREF *luprefix*

Specifies the entire ACB name (LUNAME *applid*) or a one- through five-character LUNAME prefix (LUPREF *luprefix*) that the BCI attempts to open to connect to the product region. If an open fails for any reason, interface processing terminates immediately.

When the LUPREF operand is used, a three-digit number (*nnn*) is appended to the LU prefix value. Initially *nnn* is set to 001 and an attempt to open ACB *luprefix*001 is performed. If this ACB is in use or varied inactive, the suffix keeps incrementing. The open is retried until one of the following conditions occur:

- The open is successful.
- An unrecoverable open error occurs.
- The list of defined ACBs is exhausted.

If you do *not* specify the LUNAME and LUPREF operands, then an LUPREF of NMBCI is assumed.

Examples: BCILOGON Command

```
BCILOGON NMCDRSC USER XYZ/XYZ
```

```
BCILOGON NMMAIN  USER ANET LU ABCINET PASSWORD GO
```

```
BCILOGON NMTEST  USER OPER LUPREF NMTSO PASSWORD XYZZY
```

Note: A user command exit can generate the BCILOGON command.

Other than the *applname* parameter that must be the first parameter after the BCILOGON command, subsequent parameters can be entered in any order.

BCIDISC Command—Terminate Session

The BCIDISC command terminates the session with the product region after processing with that system is complete.

The BCIDISC command has no parameters and has the following format:

```
BCIDISC
```

This command logs you off the product region and frees the VTAM ACB being used this session. If the end of the input file is reached and an active session exists, then a BCIDISC command is assumed.

BCIEXITC Command—Control Command Exit

The BCIEXITC command activates or deactivates a user command exit. You can use a user command exit program to inspect user commands before being processed by the BCI.

This command has the following format:

BCIEXITC [*exit_name*]

exit_name

Is the one- to eight-character name of the program to be loaded as the user exit for the BCI. The program must reside in a standard load library. Specifying the exit name on the BCIEXITC command loads the module only. The exit is not called until a command is next read from the input file.

If the exit name is omitted, any currently executing exit is deactivated.

Only one command exit can be active at any one time. If you must invoke a different exit, first use a BCIEXITC command to deactivate the current exit.

BCI Operation

After completing a successful BCILOGON command, BCI converses with a User Services procedure defined to the product region. The BCILOGON command makes an initial menu selection of U (for the User Services menu).

By using the optional MENU operand of the BCILOGON command, additional data can be passed to the initial procedure to invoke a special procedure for conversing with BCI. This initial procedure uses the &ZPSKIP NCL statement to receive and process this data in the usual manner.

Note: For more information, see the *Network Control Language Reference Guide*.

As previously mentioned, BCI makes use of the Virtual 3270 Interface. The User Services procedure (and its associated panel) used for the conversation must meet certain basic requirements (see page 529). The first screen line on the panel should contain three output fields for control information. The second line is an input command line, and the remaining lines are used to return any output data. The supplied procedure ($USERBCI) and panel ($USV3270) conform to these standards and should be used as a model.

BCI defines itself to the Virtual 3270 Interface as a 3270 Model-4 and, while it will only send a single command on the first input line of the panel, it will accept up to 41 data lines in the response (the Model-4 type terminal contains 43 lines of 80 characters each).

After any command has been processed by the procedure, including the BCILOGON command, the first two control fields are reported as the panel name and procedure name respectively. The third field is regarded as a return code, and set as &RETCODE. If this is non-zero, the user is disconnected and documentary messages written to the system output file. The job is canceled with a JCL return code of 12.

Procedure $USERBCI

Procedure $USERBCI contains all the commands necessary to communicate with a virtual BCI 3270. It may also be used for a real 3270 and be extended by the installation to cater for additional functions. The procedure contains comments which adequately describe the basic functions it performs.

$USERBCI recognizes several commands as functions it must execute. Any other commands received are assumed to be product commands and are processed by specifying them on an &INTCMD statement. This makes the procedure as powerful as the privileges associated with the particular user ID for which it is invoked, and can be restricted at the installation's discretion.

The results of an &INTCMD command are accumulated by issuing &INTREAD statements and placing the resulting messages on the output lines of the panel.

If all panel output lines are used, the data is sent to BCI by issuing an &PANEL statement. The messages are extracted and BCI awaits further output. More than one panel may be required to send the results of some commands. This process is automatically handled by the interface.

The following commands are not passed to the product region but are processed by the distributed procedure $USERBCI:

ECHO [nn] [text]

Is recognized by $USERBCI as a test command. The specified text is echoed nn times (up to 41 maximum) in the output lines of the panel. If nn is omitted or non-numeric, the text is echoed 3 times. If no text is supplied, a sequence of characters is returned.

EXIT

May be used to emulate the F3 key and will end procedure $USERBCI. This can be useful if $USERBCI is nested from another BCI NCL procedure. It is not used to terminate the BCI session (see page 523).

INVRC [nn]

Is used by $USERBCI to set &RETCODE to nn. If this is a non-zero value, the batch program will terminate immediately with message N96009 issued.

MORE

A request to perform another &INTREAD operation. After an &INTCMD has been issued, the results are read using &INTREAD commands. Since an unknown number of messages may be returned, the procedure makes some assumptions as to when the last message has been read. If you suspect more messages are available, these can be solicited through the MORE command. Any number of MORE commands may be issued.

NOMORE

Signifies that any outstanding messages not yet returned should be purged. This results in the procedure issuing an &INTCLEAR statement to discard any such outstanding responses.

TRACE ON|OFF

May be used to issue an NCL &CONTROL TRACE or &CONTROL NOTRACE command in the BCI NCL procedure. This may be useful for debugging BCI NCL procedures.

Customize $USERBCI

An installation may further tailor the $USERBCI procedure to extend its function. For example, certain commands could be recognized to invoke other procedures or panels if desired.

If the procedure is modified by the installation, CA recommends that you test it using a real 3270 before testing it on the BCI. The procedure can be tested at any 3270 (this does not have to be a Model-4, as the number of screen lines is handled by the procedure).

If a 132-column screen is used, output is truncated by BCI to the first 79 characters. When testing at a real terminal, MORE command processing can be abandoned by pressing F12 or F24.

For ease of testing or occasional usage, we suggest you amend the existing User Services procedure and its associated panel, to allow entry to the tailored procedure from a normal session. If you do this, then users won't have to log onto the product using a BCI user ID. This eliminates a potential conflict between BCI and online usage.

BCI Command Exit

When a record is read from the system input file, it is first scanned to determine whether it is a BCI control command. If not, it is recognized as a user command to pass to the product region. However, before it is passed, it is reformatted to remove leading blanks. If a command exit is active, it is called passing the command as a parameter. The command exit can then choose to modify the command, to reject it, or to insert additional commands into the input stream.

On entering the exit, Register 1 contains the address of a 76-byte area as follows:

Bytes 1 through 4

Contain a binary fullword indicating the length of the following data which can be a maximum of 72 bytes.

Bytes 5 through 76

Contain the user command and data with leading blanks removed and any unused characters to the right filled with blanks.

Before returning, the exit sets Register 15 with a return code to indicate what action BCI is to take with the returned data.

Return Codes

0

Indicates to pass the command to the product region. The actual command may have been modified and its length changed by updating the 76-byte area as passed as the parameter to the exit. The reformatted command is reported on the system print file (except the BCILOGON command).

4

Indicates to pass the command to the product region (as for R15=0). However, after it has been fully processed, the exit is invoked again instead of reading the next card from the system input file. Therefore, this return code is used to insert commands. The user command area presented on the subsequent invocation contains the command as returned on the last such call, and not the original command.

8

Indicates to ignore and not report the command.

The command exit can be used to generate a BCILOGON command to generate automatically user ID and password data. If a BCILOGON command is returned, only the command and the product region VTAM APPL name are reported on the system print file.

The command exit is especially useful for removing superfluous commands that, for example, a library system generates. However, any user commands before the BCI command is detected are ignored, so the BCIEXITC command itself can exist in a library member and be detected correctly.

As the command exit is loaded by the BCIEXITC command and merely executed on each subsequent command until deactivated, the program is effectively resident. Therefore, data areas can be defined within the exit to control calls to the exit during insertion processing. Additionally, if a short command is returned in the command area, the exit can use the rest of the area because that area is not inspected, amended, or reported by BCI. If the length field is greater than 72, it is assumed to contain 72. Similarly, if the length is zero or negative, it is assumed that no command is present and no action is taken. However, if the return code is 4, the exit is invoked again. In both instances, BCI does not modify the length field.

JCL Return Codes

When the BCI program terminates, it sets a JCL return code which may be tested for successful completion. The return code will have one of the following values:

0

No errors detected.

4

BCI has processed all input cards and issued Important! messages. Check the messages to ensure that BCI has made the correct assumptions.

8

BCI has processed all input cards and has found that serious errors and incorrect results are likely. Check the messages and correct the input.

12

BCI has detected serious errors processing all input cards. The results are incomplete. Check the message and correct the input.

1001-1009

The BCI NCL procedure has ended with a non-zero return code (&RETCODE). The JCL return code is the value in &RETCODE plus 1000. For example, if &RETCODE is set to 2, then the JCL return code returned will be 1002. BCI will issue message N96009 and terminate the job with the JCL return code.

U4095

BCI has detected a serious problem. The BCI output file will contain messages that indicate the nature of the error; it may be an internal error. A dump is produced which may be required for problem diagnosis by your product supplier.

Example: JCL Return Codes

```
//NMBCI JOB   etc
//NMBCI EXEC PGM=NMBCI
//SYSOUT DD SYSOUT=A
//SYSIN DD *
BCILOGON NMTEST USER TESTBCI/TESTPSWD MENU BCI LU NMDOSBCI TRANSMIT TESTFILE
SHOW USERS
LIST $USERBCI
EXEC TESTPROC
BCIDISC
BCILOGON NMACCTS PASSWORD MONEY USER WAGES
TRANSMIT PAYROLL
BCIDISC
/*
//
```

Appendix H: Virtual 3270 Interface

This section contains the following topics:

About the Virtual 3270 Interface

The Virtual 3270 Interface allows an external program to communicate with your product, in particular with NCL procedures, using full-screen dialog.

Requests are normally passed to NCL procedures running under the User Services option. The interface user program and NCL procedure can determine the state and flow of data on the conversation as they desire.

This interface may be used to connect to a product region operating in the same or a different VTAM domain.

Virtual 3270 Interface

Various calls to the interface are supported to simplify session establishment and termination, and for reading and writing the information contained in 3270 data streams.

A parameter list supplied as the DSECT $NMVTAP in the product distribution library is used for communication. Once output has been received on the session a Field Descriptor List is built and the address placed in the main parameter area. This allows the interface user program to examine the contents of the panel and to update input fields for return your product.

The NMV3270I load module can be linked to a program requiring interface services, or loaded as part of an initialization process. Calls to the interface are then supported by passing the initialized parameter area to provide the following functions:

Open

Starts a session with the product region using parameter list information to determine the target system applid, the local VTAM ACB name to be used for the session, the user ID and password for the user requesting the logon, the virtual 3270 screen size, and any initial menu selection required.

If an LUNAME prefix is used for the ACB name (APPLID) and the ACB fails to open because the APPL has been varied inactive, the operation retries using the next generic ACB name. This process is repeated until an open is successful, or until five successive inactive or unknown ACB name errors occur.

Receive

Is used to receive a full-screen of data and builds a Field Descriptor List which describes all input (and optionally output) fields found on the panel. This may then be examined by the calling application to determine the contents of the various fields.

Send

Is used to return data to the product region. The Field Descriptor List may be plugged with the address of input data which is to be placed in panel input fields. A code specifying the function key to be pressed can also be set (the default is the Enter key).

Send, then Receive

Performs both send and receive functions before returning control to the calling program. This is useful when issuing a request which expects to solicit some data in reply.

Send, then Terminate

Performs a Send then awaits session termination before returning control (for example, after a LOGOFF command has been sent). If the session does not shut down in an orderly fashion, then it is disconnected before control is returned to the calling application.

Terminate

Closes the session immediately without further I/O and returns to the caller.

More documentation may be found in the DSECT $NMVTAP describing the fields which can be set for the various calls, and those which may be set by the interface on return.

The supplied DSECT $NMFLDDS can be used to map the data returned in the Field Descriptor List obtained from the panels used in a conversation. See those DSECTS in the distributed NMMACLIB for details on this facility.

Control the Conversation

By default, NMV3270I selects the User Services menu as the initial menu selection. This can be modified on the Open call. However, no matter what function gets control and issues the first I/O, it is the responsibility of the program calling the interface and the NCL procedure (or other function) to agree to some protocol and maintain proper synchronization.

The Virtual 3270 Interface provides services which simplify VTAM session communication, and let you analyze the data being passed across the session using a 3270 full-screen conversation.

Interface to NCL Procedures

Once a session has been established through the Virtual 3270 Interface and has made an initial menu selection that passes control to an NCL procedure, the procedure begins the conversation by issuing an &PANEL statement. This results in a 3270 data stream being sent to the interface.

The interface user must issue a Receive call at which time the virtual panel image and Field Descriptor List are built and returned to the caller.

By requesting a Send (or a Send then Receive) function, any input is returned with the appropriate attention key set. This satisfies the &PANEL statement at which time the procedure can examine the input fields through their token names, and the key through the &INKEY system variable in the usual manner.

Panel Characteristics

While any panel definition could be used for this conversation, the simpler the panel definition the easier it is for the interface user to process. Panel $USV3270 is used by both the Batch Command Interface and the TSO Command Interface and uses specific conventions.

The panel is defined as a Model-4 (43 line) panel, the first line containing three protected fields as follows:

- The first field contains the characters $USV3270 and can be used to verify that the interface user is connected to an NCL procedure designed to handle this type of communication.
- The second field contains the system variable &0. On output this will be substituted and contains the name of the currently executing procedure. This helps maintain synchronization throughout the conversation.
- The third field contains the system variable &RETCODE which can be used to determine the status of processing within the procedure.

The contents of these fields are extracted and placed into defined areas of the parameter list passed between the user program and the interface, regardless of whether the user requested all or only input fields be built in the Field Descriptor List. Hence there is an advantage in maintaining this first line convention for any application.

The second line of panel $USV3270 is an input field and contains the token name &REQUEST. This of course appears to the user program as the first input field in the Field Descriptor List and hence is easily accessible. Normally this would be used to describe the type of function that the interface user is requesting from the NCL procedure.

Subsequent lines of the panel are all input lines and may be used to contain qualifying input from the user application, or for returned text data from the NCL procedure.

Where any external application is required to use the services of the Virtual 3270 Interface, CA recommends that the procedures and panels described be used as models to assist development.

Index