System Software Library

Distributed Systems Network Management (DSNM) Subsystem Interface Development Guide

Abstract

This manual describes the Distributed Systems Network Management (DSNM) services that support network management applications. It describes how to use the program frame and library services supplied by Tandem for the development of the subsystem interface processes that integrate additional subsystems into the base of DSNM-managed subsystems.

Product Version

DSNM D30

Supported Releases

This manual supports D30.01 and all subsequent releases until otherwise indicated in a new edition

Part Number	Edition	Published	Release ID
109759	Second	February 1996	D30.03

Document History

Edition	Part Number	Product Version	Earliest Supported Release	Published
First	029783	DSMS C21	C20	December 1990
Second	109759	DSNM D30	D30.01	February 1996

New editions incorporate any updates since the previous edition.

A plus sign (+) after a release ID indicates that this manual describes function added to the base release, either by an interim product modification (IPM) or by a new product version on a .99 site update tape (SUT).

Ordering Information

For manual ordering information: domestic U.S. customers, call 1-800-243-6886; international customers, contact your local sales representative.

Document Disclaimer

Information contained in a manual is subject to change without notice. Please check with your authorized Tandem representative to make sure you have the most recent information.

Export Statement

Export of the information contained in this manual may require authorization from the U.S. Department of Commerce.

Examples

Examples and sample programs are for illustration only and may not be suited for your particular purpose. Tandem does not warrant, guarantee, or make any representations regarding the use or the results of the use of any examples or sample programs in any documentation. You should verify the applicability of any example or sample program before placing the software into productive use.

U.S. Government Customers

FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE:

These notices shall be marked on any reproduction of this data, in whole or in part.

NOTICE: Notwithstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software—Restricted Rights clause.

RESTRICTED RIGHTS NOTICE: Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.

RESTRICTED RIGHTS LEGEND: Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraphþ(b)(3)(B) of the rights in Technical Data and Computer Software clause in DAR 7-104.9(a). This computer software is submitted with “restricted rights.” Use, duplication or disclosure is subject to the restrictions as set forth in NASA FARþSUP 18-52þ227-79 (Aprilþ1985) “Commercial Computer Software—Restricted Rights (Aprilþ1985).” If the contract contains the Clause at 18-52þ227-74 “Rights in Data General” then the “Alternate III” clause applies.

U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract.

New and Changed Information

This edition of the *Distributed Systems Network Management (DSNM) Subsystem Interface Development Guide* has been updated to include functions and features added to DSNM for the C31 and D30 releases of the product.

The operating system for Tandem NonStop systems, formerly called the Guardian operating system, is now called the Tandem NonStop Kernel. This change reflects Tandem's current and future operating system enhancements that further enable open systems and application portability.

The major changes to each section are as follows:

- Section 1, "Overview of DSNM," has been expanded to include new components such as the conversational interface process (CIP), and the list of supported products has been expanded to include NonStop NET/MASTER Management Services (MS). Section 1 now includes information on installation, process configuration, running more than one copy of DSNM concurrently, and mixed network requirements.
- Section 2, "DSNM Commands," has been expanded to include the INQUIRE and UPDATE commands.
- In Section 3, "I Process Development Process," references to _COMMAND^CONTEXT^ADDRESS have been replaced with _THREAD^CONTEXT^ADDRESS and minor changes have been made.
- In Section 4, "DSNM Command Requirements," the DSNM command requirements have been expanded to include INQUIRE and UPDATE.
- The configuration and process parameter descriptions in Section 5, "DSNM Process Startup Functions," have been updated, expanded, and reformatted.
- In Section 6, "Configuring a New Subsystem Into DSNM," changes to the DSNM configuration files and parameters are reflected where necessary.
- Section 7, "DSNMCom: The I Process Test Utility," documents the new DSNMCom commands and parameters.
- New DSNM error codes have been added to Appendix B, "DSNM Error Codes."
- New DSNM SPI components have been added to Appendix C, "Data Definition Language (DDL)-Defined DSNM SPI Components."

Contents

3. I Process Development Process

A. DSNM Library Services

B. DSNM Error Codes

C. Data Definition Language (DDL)-Defined DSNM SPI Components

D. Sample I Process Program Code

Index

Figures

Tables

About This Manual

The *Distributed Systems Network Management (DSNM) Subsystem Interface Development Guide* is written for programmers who develop the interface processes (I processes) that allow subsystems or applications to be managed by network management products that Distributed Systems Network Management (DSNM) services support.

DSNM services support network management applications that automate and simplify the management of Tandem systems and networks. This manual describes a program frame, library services, and a detailed development model that facilitate the development of the interface processes between the targeted subsystems and the DSNM services layer within a network management application architecture.

The DSNM product is part of the Distributed Systems Management Solutions (DSMS) package, which provides the capability of monitoring and managing a single Tandem node or a network of Tandem systems from a single terminal. The *Distributed Systems Management Solutions (DSMS) System Management Guide* is a companion manual to the *DSNM Subsystem Interface Development Guide*.

How This Manual Is Organized

This manual serves as both a reference manual and a programmer's guide. It is organized as follows:

- Section 1, “Overview of DSNM,” provides a functional overview of DSNM, and explains how network management applications interact with the underlying DSNM service layer processes and components to control and monitor objects.
- Section 2, “DSNM Commands,” describes the DSNM commands and provides syntax descriptions in sufficient detail for testing I program code in an end-user capacity.
- Section 3, “I Process Development Process,” introduces the conceptual model upon which the program frame and library services are based. It also provides a detailed development model and associated rules for using the I process development software correctly and effectively.
- Section 4, “DSNM Command Requirements,” defines the DSNM requirements for carrying out each supported DSNM operation.
- Section 5, “DSNM Process Startup Functions,” describes the library services that take advantage of the expanded scope of the DSNM configuration file(s) to perform process startup and subsystem configuration parameter retrieval.
- Section 6, “Configuring a New Subsystem Into DSNM,” documents the steps necessary to configure a new subsystem into DSNM.
- Section 7, “DSNMCom: The I Process Test Utility,” describes how to use DSNMCom, the I process test utility.

- Appendix A, “DSNM Library Services,” describes the syntax and parameters (as applicable) for each procedure call, define, literal, global variable, and structure template.
- Appendix B, “DSNM Error Codes,” defines the ZDSN error codes.
- Appendix C, “Data Definition Language (DDL)-Defined DSNM SPI Components,” lists the Subsystem Programmatic Interface (SPI) Data Definition Language (DDL) constant and structure definitions for user-written procedures.
- Appendix D, “Sample I Process Program Code,” provides a sample I process program, illustrating the program model and associated library services.

Where to Go for More Information

If you are writing an interface for an existing Tandem subsystem, you need the documentation for the product you intend to manage with DSNM.

Although the purpose of the interface development software is to create interface processes that make the Tandem Subsystem Programmatic Interface (SPI) protocol used by DSNM transparent to your subsystem, you may also want to refer to the following manuals for more information about the Distributed Systems Management (DSM) architecture upon which DSNM is built:

- *Introduction to Distributed Systems Management (DSM)*, which provides an overview of DSM and its components. DSM products support the management of system and network resources and operations.
- *SPI Programming Manual*, which describes the operating system procedures that programmers call to process Subsystem Programmatic Interface (SPI) messages. The manual also presents conventions that regulate message content and interpretation, provides programming guidelines and examples, and describes the common ZSPI data definitions.
- *SPI Common Extensions Manual*, which describes conventions that extend the basic SPI interface, as described in the *SPI Programming Manual.*
- *EMS Manual*, which describes the Event Management Service (EMS). EMS is a collection of processes, tools, and interfaces that provide event-message collection and distribution in the DSM environment.
- *Distributed Name Service (DNS) Management Operations Manual*, which describes the interactive DNS interface DSNCOM, used to maintain a database of object names controlled by Tandem and other systems.
- *NonStop NET/MASTER MS System Management Guide*, which describes the NonStop NET/MASTER MS configuration and security management processes, and the specific tasks required to configure and secure NonStop NET/MASTER MS.
- *NonStop NET/MASTER MS Operator’s Guide*, which describes how to use the various components of NonStop NET/MASTER MS to perform system and network management tasks.

- *NonStop TS/MP and Pathway System Management Guide*, which provides guidelines for configuring and controlling Pathway transaction processing systems.

The following manuals provide information about the DSMS network management products that currently use the DSNM services layer:

- *Distributed Systems Management Solutions (DSMS) System Management Guide*, which provides information for installing and managing DSNM, NetCommand, and NetStatus software in both DSMS and NonStop NET/MASTER MS operations environments.
- *User's Guide to DSNM Commands*, which discusses the syntax and use of the DSNM commands.
- *NetStatus User's Guide*, which explains both usage and management of the NetStatus monitoring software.

Your Comments Invited

After using this manual, please take a moment to send us your comments. You can do this by returning a Reader Comment Card or by sending an Internet mail message.

A Reader Comment Card is located at the back of printed manuals and as a separate file on the Tandem CD Read disc. You can either fax or mail the card to us. The fax number and mailing address are provided on the card.

Also provided on the Reader Comment Card is an Internet mail address. When you send an Internet mail message to us, we immediately acknowledge receipt of your message. A detailed response to your message is sent as soon as possible. Be sure to include your name, company name, address, and phone number in your message. If your comments are specific to a particular manual, also include the part number and title of the manual.

Many of the improvements you see in Tandem manuals are a result of suggestions from our customers. Please take this opportunity to help us improve future manuals.

Notation Conventions

General Syntax Notation

The following list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name
```

```
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list may be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
LIGHTS [ ON              ]
       [ OFF             ]
       [ SMOOTH [ num ] ]
```

```
K [ X | D ] address-1
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list may be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }
```

```
ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address-1 [ , new-value ]...
```

```
[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be entered as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must enter as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In the following example, there are no spaces permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] CONTROLLER

   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                      !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length           !i:i
                            , filename2:length ) ;       !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                          !i
                        , [ filename:maxlen ] ) ;          !o:i
```

Notation for Messages

The following list summarizes the notation conventions for the presentation of displayed messages in this manual.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

```
p-register
```

```
process-name
```

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number = number [ Subject = first-subject-value ]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list might be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
LDEV ldev [ CU %ccu | CU %... ] UP [ (cpu,chan,%ctlr,%unit) ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list might be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LBU { X | Y } POWER FAIL
```

```
process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown.          }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The %þnotation precedes an octal number. The %Bþnotation precedes a binary number. The %Hþnotation precedes a hexadecimal number. For example:

```
%005400
```

```
P=%p-register E=%e-register
```

Notation for Management Programming Interfaces

UPPERCASE LETTERS. Uppercase letters indicate names from definition files; enter these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters. Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

!r. The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME       token-type ZSPI-TYP-STRING.          !r
```

!o. The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER       token-type ZSPI-TYP-FNAME32.         !o
```

Change Bar Notation

Change bars are used to indicate substantive differences between this edition of the manual and the preceding edition. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

1 Overview of DSNM

Scope of This Section

This section provides an overview of the Distributed Systems Network Management (DSNM) services, which collate information from multiple subsystems and provide a consistent view between network-management applications and the diverse subsystems being managed.

What is DSNM?

The DSNM product provides a management service layer between management applications and individual management facilities for Tandem subsystems and user applications. DSNM works with the DSM products to present network-management applications with a uniform interface to Tandem subsystems and applications. It collates information from multiple subsystems and provides a consistent view between the operations environment and the diverse subsystems managed by various network-management products. DSNM provides the following services and operations:

- Maintains a real-time database about subsystem objects defined to nodes in the network.
- Processes a set of control, information, and update commands that it receives from network-management applications.
- Translates command responses from different subsystems into standard DSNM responses.
- Interprets subsystem events and forwards object state change information to the requesting program.

NetCommand and NetStatus are complementary management applications that present the major DSNM services to a human network operator and provide additional functions of their own.

Note. In the context of Tandem systems, a subsystem is a process or set of processes that manages a cohesive set of objects. Objects are items subject to independent reference and control by a subsystem: for example, PATHMON-controlled applications and terminals handled by a PATHMON process, communication lines controlled by a SNAX line-handler process, or jobs managed by the spooler. Objects relate conceptually to the subsystems that control them, and are often referred to as ”subsystem objects.”

Applications Supported by DSNM

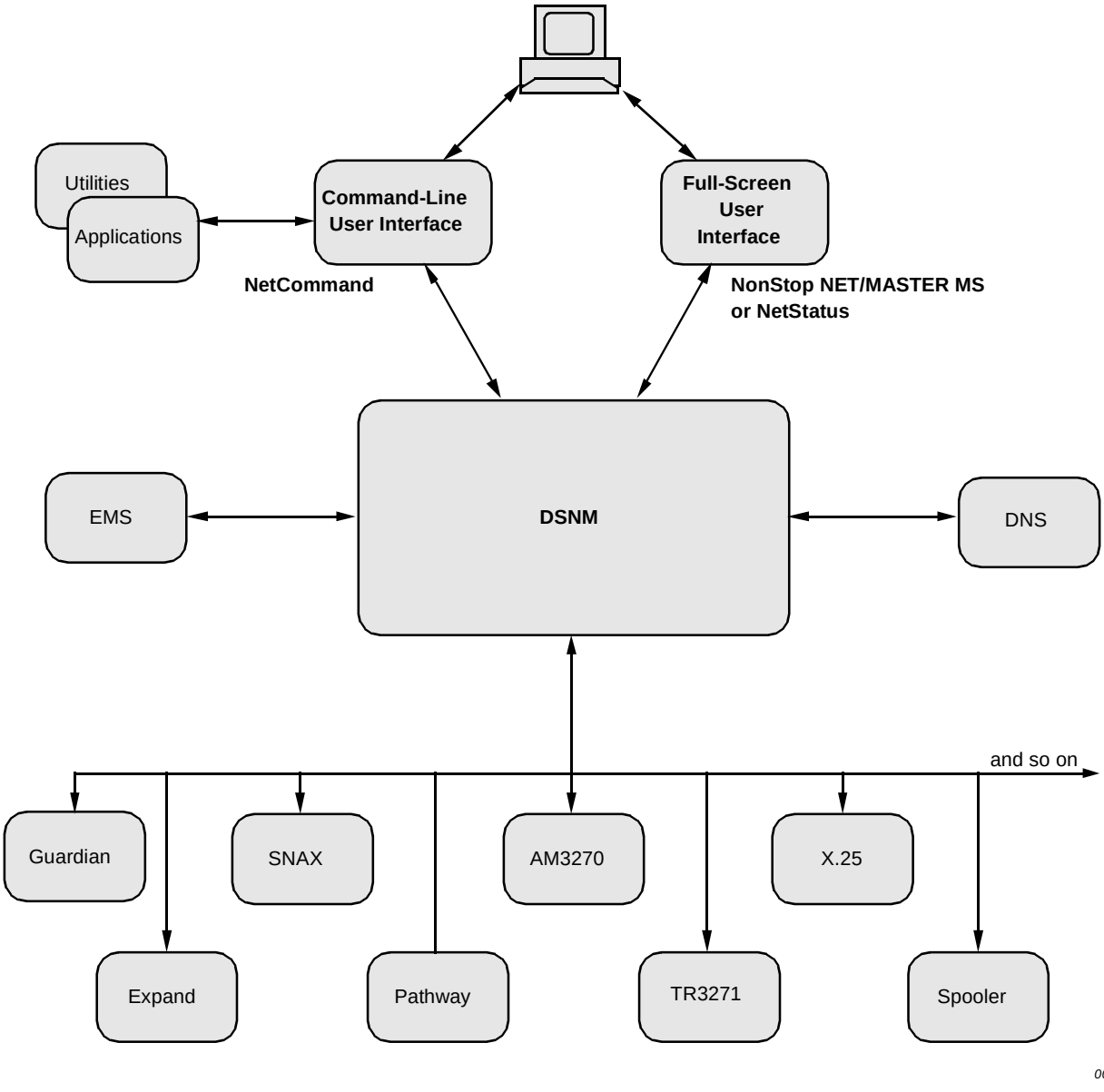
DSNM supports three network-management products—NonStop NET/MASTER MS, NetCommand, and NetStatus. Figure 1-1 illustrates the relationship between the user interface (NonStop NET/MASTER MS, NetCommand, or NetStatus), the DSNM services layer, and the subsystems being managed.

Collectively, the NetCommand and NetStatus applications, along with the DSNM services layer, compose supported networks and applications. The DSMS products

address day-to-day network-management issues. The DSMS products are primarily oriented toward early fault detection and recovery.

NonStop NET/MASTER MS provides comprehensive collections of network-management services, such as automated operations, capacity and change management, configuration management, problem management, and performance management. DSMS products are used by and with NonStop NET/MASTER MS. In particular, the DSNM command infrastructure provides the command interface to Tandem subsystems for NonStop NET/MASTER MS.

Figure 1-1. Network-Management Application Components



NonStop NET/MASTER MS

NonStop NET/MASTER MS is a network-management product that allows you to monitor and manage a single Tandem system or an entire network from a single terminal. Having NonStop NET/MASTER MS installed on your system also allows your local system to be monitored and/or managed remotely as part of a network of systems managed by NonStop NET/MASTER MS, SOLVE management services, and NetView products.

With NonStop NET/MASTER MS, you can:

- View event messages generated by both local and remote systems throughout a network.
- Issue commands to remotely control and gather information about any peer system in the network, and have the responses displayed on your local terminal.
- Run Tandem NonStop Kernel utilities, TACL routines, and other external conversational-mode utilities, and control Tandem block-mode applications.
- Write and execute custom applications and operations management automation procedures with NonStop NET/MASTER Network Control Language (NCL), a high-level language for automating system and network-management tasks.
- Browse activity log files, where messages arriving at your local NonStop NET/MASTER MS system are logged.

NetCommand

The NetCommand management application is a conversational user interface to DSNM, primarily for command and control operations. NetCommand provides a secure TACL environment for operational control of subsystems throughout a distributed network.

As the manager of the system, you can define operator profiles and exercise control over operator capabilities. NetCommand is extensible by using user-written TACL routines and macros; these can be entered into any operator's command set.

NetCommand allows an authorized operator direct access to the DSNM command set, as well as access to individual subsystem command interfaces or other utilities, such as the Peripheral Utility Program (PUP) or the Subsystem Control Facility (SCF). It also maintains an audit trail of operator commands and command responses through every operator session.

With the DSNM command set, you can:

- Control various subsystems in a network by removing resources from service and restoring them later (ABORT, STOP, START).
- Display status information about network objects (AGGREGATE, INQUIRE, STATUS).
- Gather information about how network objects are configured (INFO).

- Display operational statistics about network objects (STATISTICS).
- Define how the DSNM services layer monitors subsystem objects (UPDATE).

Use of NetCommand is discussed in the *User's Guide to DSNM Commands*.

NetStatus

The NetStatus management application is a Pathway-environment application that uses the monitoring facility of DSNM to provide a full-screen status display, which is continuously updated. You can view the status of objects, a system, or an entire network of systems from any point in the network. From the status display, you can use function keys to perform a variety of operations: navigate, execute DSNM commands, query, and control components. Frequently, you can correct an identified problem with a single keystroke. NetStatus also allows access to the NetCommand conversational interface directly from the block-mode Pathway screen.

The NetStatus operations environment is described in the *NetStatus User's Guide*.

The Network-Management Architecture

A network-management architecture is generally divided into three layers:

- Operations layer
- Management services layer
- Subsystem layer

Each layer is a set of related or parallel services having a well-defined interface with the other layers.

The Operations Layer

The operations layer provides the interface for human operators. It may consist of command-line and full-screen user interfaces. NonStop NET/MASTER MS, NetCommand, and NetStatus are examples of user interfaces to DSNM.

NonStop NET/MASTER MS

NonStop NET/MASTER MS is network and system management product. It is composed of the following major services:

- Operator Control Services (OCS), which provides the central point of operational control of your local Tandem system, your local NonStop NET/MASTER MS system, and remote systems. OCS provides a command input line to enter NonStop NET/MASTER MS commands.
- Edit Services, which provides access to the Tandem text editor, PS Text Edit (TEDIT). Edit Services allows NCL programmers to create and check NCL procedures and panel description files.

- User ID Management Services (UMS), a security service that enables the definition of authorized NonStop NET/MASTER MS users and their associated functions and privileges.
- Inter-NET/MASTER Connection (INMC), which allows multiple NonStop NET/MASTER MS and SOLVE management services systems to be connected and controlled from one location.
- Remote Operator Control (ROC), which allows users to log on from a local NonStop NET/MASTER MS system to a remote NonStop NET/MASTER MS or SOLVE management services system, to execute commands on the remote system, and to receive the results at the local NonStop NET/MASTER MS system.
- Inter-System Routing (ISR), which enables, disables, and controls system-level message flows between multiple NonStop NET/MASTER MS and SOLVE management services systems.

NetCommand

NetCommand is a command-line interface to DSNM. NetCommand is composed of the following:

- The NetCommand configuration file (NETCONF), which contains the following information about the NetCommand environment:
  - The log file and limit on the length of log entries.
  - The name of the DSNM command server process (discussed under “The Management Services Layer” on page 1-8).
  - The name of the NetStatus terminal start server process (discussed under “NetStatus” on page 1-4).
  - The command sets for specified groups of users and terminals.
  - The node access restrictions for specified groups of users and terminals.
  - Default NetStatus display sets for specified groups of users and terminals.
- A TACL requester process (NETCMD), which:
  - Defines the user environment (such as the operator’s command set, what nodes the operator can control, and the number of response lines logged).
  - Secures DSNM by checking each command against the user environment.
  - Executes TACL commands entered by the user or received from NetStatus.
  - Passes DSNM commands to the NetCommand server process.
- TAL-based NetCommand server process (NETSVR), which:
  - Parse DSNM commands into tokenized form.
  - Pass control and status commands to DSNM.
  - Invoke utilities.

- Format responses to DSNM commands for the NetCommand requester process.
- Send display set information to NetStatus.
- Execute non-DSNM commands for NetStatus.
- Log commands and responses.

The *Distributed Systems Management Solutions (DSMS) System Management Guide* contains additional information on NetCommand components.

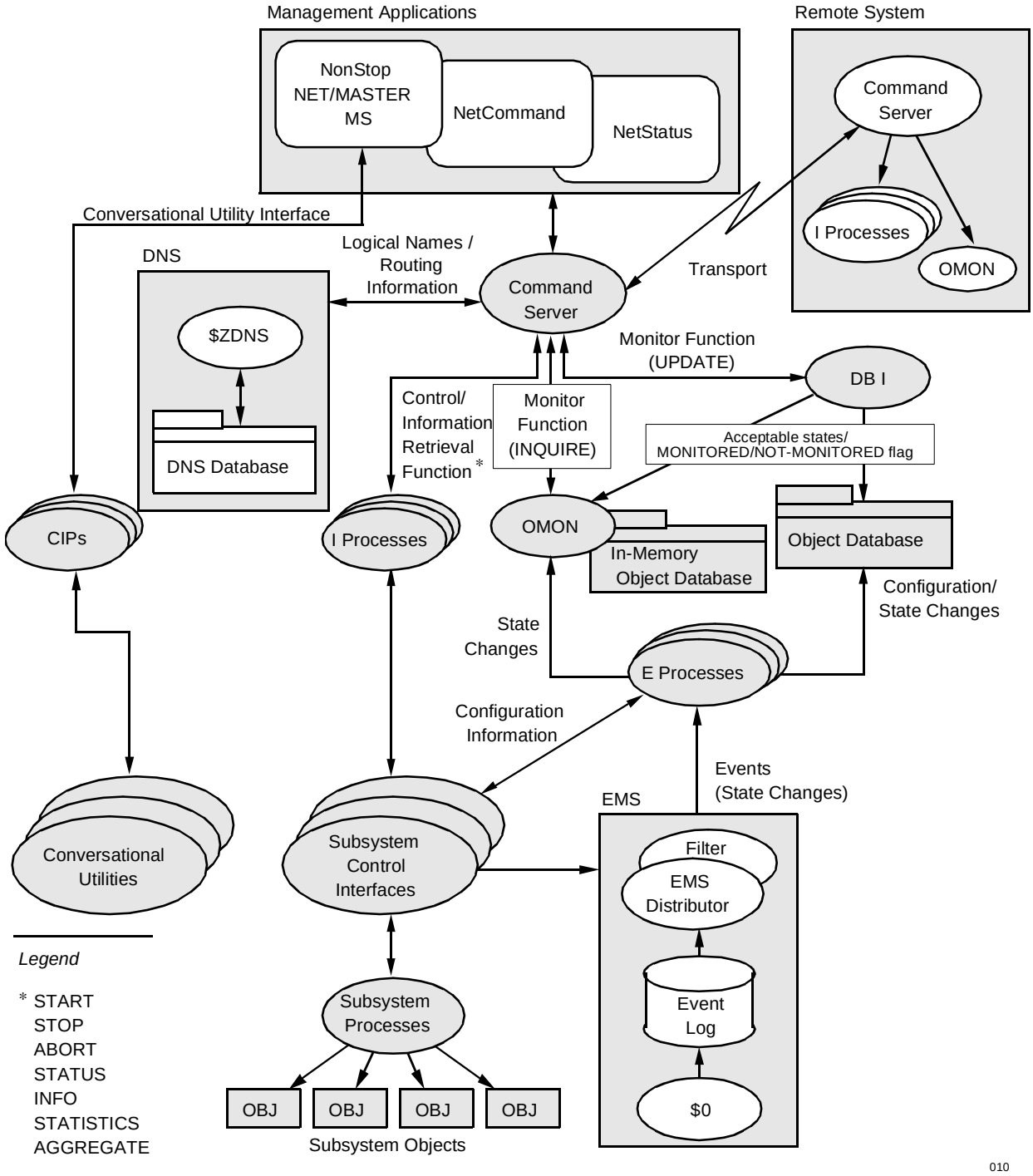
NetStatus

NetStatus is a full-screen interface to DSNM. NetStatus runs in a PATHMON environment composed of:

- A terminal start server (TERM-START-SVR), which:
  - Initiates the terminal session.
  - Retrieves information identifying the network objects to be monitored and controlled and sends this information to the screen requester for display.
  - Allocates and deallocates NetStatus threads (consisting of a NetStatus terminal and its associated NetStatus server class as defined in DSMS PATHWAY) when a user enters and exits NetStatus.
- A SCREEN COBOL screen requester, which:
  - Displays object status information, help screens, and text response screens on the operator's terminal.
  - Passes commands to the NetStatus servers.
- NetStatus servers (NETSTATUS-SVR), which provide status, command, page control, response formatting, notification, and command and response logging services for NetStatus. NetStatus servers:
  - Secure DSNM by checking each command against the user environment.
  - Parse DSNM commands into tokenized form.
  - Send control and status commands to and receive responses from the DSNM command server.
  - Deliver object state change information forwarded by the command server to the designated screen requester for display.
  - Send TACL commands to NetCommand.

The *Distributed Systems Management Solutions (DSMS) System Management Guide* contains additional information on NetStatus components.

Figure 1-2. DSNM and DSM Functional Connections



The Management Services Layer

The management services layer performs services for the operations layer. It provides for object resolution, the routing of commands to appropriate nodes and subsystems, and the handling of event messages. Tandem's Distributed Systems Management (DSM) services contribute object resolution and event management facilities. Figure 1-2 displays the DSNM and DSM functional connections.

DSM Components

- Distributed name service (DNS)

  DNS is a subsystem that manages a distributed database of names, aliases, groups, and composites of system and network objects. The DSNM command server (discussed later in this subsection) communicates with the DNS name manager process ($ZDNS) to resolve aliases, groups, and composite names into their constituent subsystem-defined object names. DNS is described in the *Distributed Name Service (DNS) Management Operations Manual*.

- Event management service (EMS)

  EMS provides event-collection, logging, and distribution facilities:

  - The EMS collector process ($0) receives event messages from the processes that control subsystem devices and writes them to an event log.
  - An EMS distributor process reads event messages from the event log, filters out the messages for each DSNM-managed subsystem, and forwards them to the appropriate event monitoring process (“E process,” discussed later in this subsection). The locations of EMS filter files are defined to DSNM as part of the installation procedure.

  EMS is described in the *EMS Manual*.

DSNM Components

- The command server process, which performs the following operations:
  - Receives commands from server processes.
  - Communicates with the DNS name manager process to resolve object names into subsystem object names.
  - Sends command and object information to the interface process (I process) associated with the targeted subsystem or to the object monitor process (discussed next).
  - Collects response information from the I process or the object monitor process and forwards it to the server process that initiated the command.

- The object monitor process (OMON), which performs the following
  - Receives state change information from the event monitoring processes (E processes, discussed later in this subsection).
  - Creates and maintains an in-memory database of current object state information.
  - Responds to command server requests for object state information.
  - Receives update changes from the database interface process (discussed later in this subsection).
- The object database, which stores the following information for each object:
  - Whether the object is currently monitored.
  - A description of the object, including its subsystem, object type, manager, and parent within the subsystem hierarchy.
  - Current subsystem status.
  - Current high-level status (UP, DOWN, or PENDING).
  - Acceptable states.

  For each subsystem, the object database contains the following information:
  - The status of the subsystem.
  - Whether the subsystem has been acquired by the object monitor (OMON); this affects what the E processes must do when a state change occurs.

  The object database is originally configured by the E process for each supported subsystem.
- The database interface process (DBI), which performs the following:
  - Updates entries in the object database in response to requests from the command server regarding which objects are to be monitored, how much error information is to be displayed, and the criteria by which data is highlighted on the screen.
  - Reports these types of update changes to the object monitor process.
- A subsystem interface process (I process) for each supported subsystem provides the interface between the targeted subsystem and the DSNM command processing services. The I processes:
  - Convert DSNM commands affecting subsystem objects into a sequence of syntactically correct subsystem-specific commands.
  - Pass commands to the subsystem’s control interface process, which is responsible for executing the commands.
  - Translate subsystem responses and return them to the command server process.

- The conversational interface processes (CIP), which provides access to conversational utilities. It creates and terminates utility processes and emulates a terminal from the utility's point of view.
- An event monitoring process (E process) for each supported subsystem provides the interface between the targeted subsystem and the DSNM object monitoring services. The E processes:
  - Forward state change information from the EMS distributor process to the object monitor process.
  - Update state change information and other operational statistics in the object database.
  - Rebuild the object data base after system reconfiguration.

The Subsystem Layer

The subsystem layer, an example of which is shown in Figure 1-3, comprises the subsystems managed by the network-management application. This layer includes the subsystem control interface processes (CIs) and the subsystem resources.

Control Interface Processes (CIs)

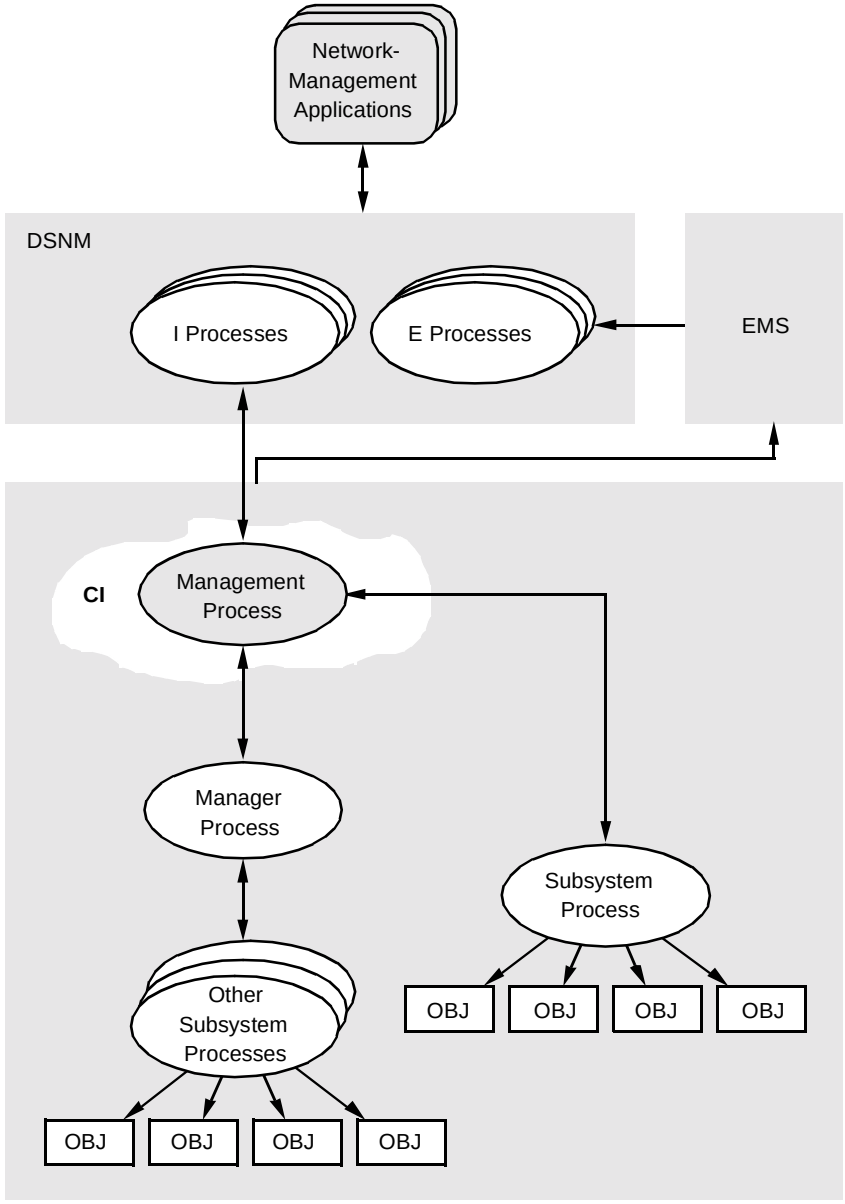
Subsystems support control and inquiry through their CI processes. In this manual, the term "CI" is used in the general sense, to mean any gateway to the subsystem for control and inquiry.

A CI is typically a management process such as PATHMON (the NonStop TS/MP control process); a public interface management process such as the SCP communications control facility, which further communicates with a private or privileged subsystem manager process to actually execute commands; or possibly a set of procedure calls such as are available for the Spooler.

A CI may execute as a server such as PATHMON or SCP, or as a requester such as PUP. Requester CIs generally have a textual interface; servers variously use text, formatted messages, or the subsystem programmatic interface (SPI).

Note. This release of the DSNM subsystem interface development software addresses server-type CIs only.

Figure 1-3. The Subsystem Layer



Subsystem Objects

A subsystem object is an entity subject to independent reference or control by a subsystem CI process. Examples of subsystem objects include:

- SNAX lines, physical units, and logical units
- AM3270 lines and subdevices
- TR3271 lines and subdevices
- X25AM lines and subdevices

- PATHMON-controlled terminal control processes, terminals, and server classes
- Expand paths and lines
- Tandem NonStop Kernel disks and processors
- Spooler devices

See the *Tandem NonStop Kernel Documents* (softdocs) for a complete list of the subsystems supported by the current level of your DSNM software.

Installing DSMS Products

DSMS software arrives at your site on a site update tape (SUT). You install DSMS software from the SUT onto your local node by running the system installation program, Install, as directed in the *INSTALL User's Guide*. Be certain to run the full system Install, including the REPSUBSYS and SYSGEN phases.

The NonStop Kernel Install program places most of the DSNM files in the DSMS installation subvolume ZDSMS. The default DSNM configuration file ZDSNCONF is installed in $SYSTEM.SYSTEM.

You may choose to copy the ZDSMS files (except the default configuration file, ZDSNCONF) to a working subvolume or to distribute them over several subvolumes. If you distribute files over several subvolumes, certain files must be placed in the same subvolume for proper operation.

Refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide* for more information about working with configuration files, including steps to customize your DSNM environment.

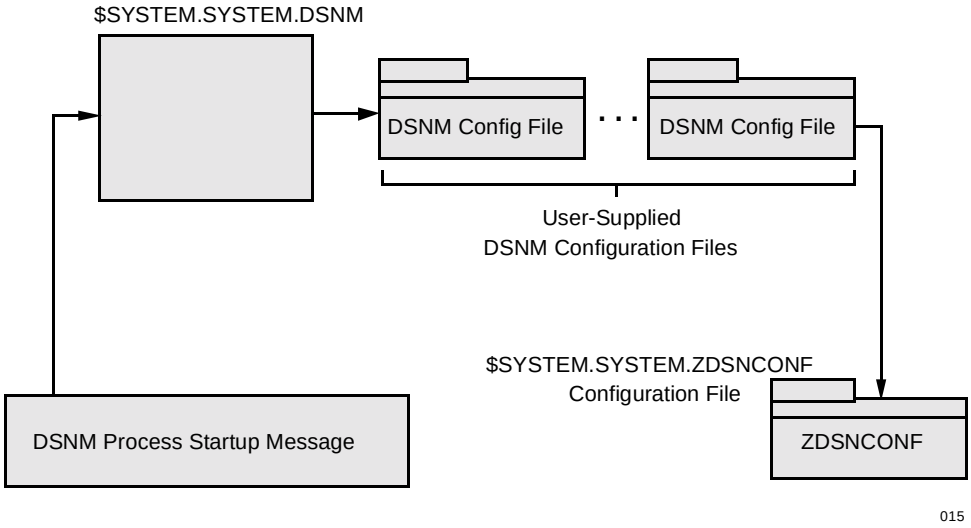
Startup Sequence and Configuration Files

A major component of DSNM is its process configuration. There are four elements to DSNM process configuration:

- The process startup message (the SERVER class STARTUP attribute in the DSNM Pathway configuration file)
- The $SYSTEM.SYSTEM.DSNM file
- User-supplied DSNM configuration file(s)
- The $SYSTEM.SYSTEM.ZDSNCONF file delivered with DSNM

Figure 1-4 shows the startup sequence that each DSNM process goes through to establish its running configuration. Section 6, "Configuring a New Subsystem Into DSNM," provides more information on the DSNM configuration files.

Figure 1-4. DSNM Process Startup and Configuration Components



Running DSNM Products

You can run DSMS products in a default configuration after the REPSUBSYS phase of the Install program in the following operations environments:

- DSNM and NetCommand in a DSMS operations environment
- DSNM and NetCommand with NetStatus in a DSMS operations environment
- DSNM alone in a NonStop NET/MASTER MS operations environment
- DSNM with NetStatus in a NonStop NET/MASTER MS operations environment
- DSNM started externally, with or without NetStatus, in a NonStop NET/MASTER MS operations environment

Refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide* for a discussion of each of these environments.

Installing More Than One Copy of DSNM Concurrently

Just as the DSNM product runs in an operations environment such as DSMS or NonStop NET/MASTER MS, every DSNM process runs in a DSNM environment. A DSNM environment defines configuration characteristics of the process. You can define any number of DSNM environments, each of which may be run in a default configuration or may be customized to any degree desired. For more information about customizing a DSNM environment, refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide*.

You can have several copies of the same DSNM environment active on a Tandem node. For example, you may want to use one DSNM copy for operations and another for testing.

To run concurrent copies with no DSNM configuration, use the standard names and change only the process prefix character.

If you are running DSNM under NonStop NET/MASTER MS, no special configuration is required. DSNM processes take their process prefix character from the first character of the NonStop NET/MASTER NCP process (refer to the *NonStop NET/MASTER MS System Management Guide* for more information).

If you are running DSNM as a PATHMON-controlled application, you must create a separate PATHMON configuration for each concurrent copy of DSNM you wish to run.

Refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide* for more information on running multiple copies of DSNM.

Mixed Network Requirements

DSNM can operate in a network that includes systems running both C-series and D-series versions of the Tandem Nonstop operating system. In such a mixed network, the DSNM modules that access remote files or processes must run at low PINs (processor identification numbers). All DSNM modules are released with the HIGHPIN parameter set to OFF.

Refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide* and the documentation that accompanies your site update tape (SUT) for information on system configuration and these DSNM modules.

Extending DSNM Support

Suitable subsystems can be added to the existing base of DSNM-managed subsystems in a modular way, without modifying existing subsystem support, by providing an I process and an E process for the targeted subsystem.

The remainder of this manual describes the development of I processes.

2 DSNM Commands

Scope of This Section

This section describes the DSNM commands that your subsystem interface process (I process) must support. It provides syntax descriptions in sufficient detail to allow you to test the commands in an end-user's capacity, explains the effect of the valid modifiers on each command, and defines what the outcome of each command should be. For complete descriptions and syntax of all DSNM commands, see the *User's Guide to DSNM Commands*.

Command Line Syntax

All DSNM commands, except AGGREGATE, have the same general syntax, which includes the following information:

- Command
- Object specification
- Modifiers
- Parameters

The DSNM command syntax is shown below:

```
command objectspec [, objectspec ][, modifier ][, parameter ]
```

Commands

command can be any one of the following:

- ABORT
- INFO
- INQUIRE
- START
- STATISTICS
- STATUS
- STOP
- UPDATE

Object Specification

An object specification gives information that DSNM uses to identify objects against which a command is issued. If there is more than one object specification, each must be separated from the other by a comma.

An object specification must include the name of at least one object. It can also include optional qualifiers (which help identify the objects you are specifying) and a hierarchy modifier (which determines—based on the subsystem object hierarchy—the objects to be included).

The syntax of the object specification is shown in the following box:

```
[subsys] [type] [\node.]name [[\node.]name] ...
   [UNDER [\node.]$manager] [hierarchy-modifier]
```

subsys

is a qualifier that identifies the name of the subsystem that controls the objects you are specifying. If you do not specify a subsystem, DSNM attempts to determine the objects' subsystems from the operating system, Distributed Name Service (DNS), and the rest of the object specification.

type

is a qualifier that identifies the type of the object you are specifying. The object type must be valid in the specified subsystem. If you do not specify an object type, DSNM attempts to determine the object types from the operating system, DNS, and the rest of the object specification.

[*node*.]*name*

is one of the following:

- A subsystem object name.
- An alias for a subsystem object name defined in the DNS database.
- A group name defined in the DNS database.
- A composite name defined in the DNS database.
- A wild-card (*) specification, if permitted by the subsystem. (The wild card is combined with the qualifiers to specify all objects that belong to the same subsystem, object type, node, and—if applicable—subsystem manager.)

`[\node.]$manager]`

is a qualifier that identifies the name of the manager process for the objects you are specifying. (Specify it only if it is applicable to the specified subsystem.)

Note. You must specify the node if it is not the local node and if any of the following are true:

- The name is a wild card(*).
- The name begins with a dollar sign ($).
- The name includes a manager.
- The name is not in the DNS database.

If the name is an alias, a group, or a composite, you can omit the node. DSNM determines the node from the DNS database.

---

hierarchy-modifier

determines which objects are to be included, based on the subsystem object hierarchy:

ALL	Causes the command to affect both the specified objects and their subordinate objects. This is the default value.
ONLY	Causes the command to affect the specified objects, but not the objects subordinate to them.
SUBONLY	Causes the command to affect the objects subordinate to the specified objects, but not the specified objects themselves.

Modifiers

Modifiers qualify the scope or output of the command. Possible modifiers are:

- Hierarchy modifier—determines, on the basis of the subsystem object hierarchy, which objects are to be affected by the command. The hierarchy modifier values are:

ALL	Applies the command to the object itself and to all subordinate subsystem objects. This is the default value.
ONLY	Applies the command to only the specified object(s).
SUBONLY	Applies the command to the subordinate subsystem objects only, but not the specified objects themselves.

- Error modifier—determines how much information is reported when the command is correct, but the objects against which the command is being executed produce errors. The error modifier values are:

ERROR-BRIEF	Returns a single line of error text. This is the default value.
ERROR-DETAIL	Returns all available error information.
ERROR-SUPPRESS	Returns no error information.

- State modifier—restricts the scope of the command to a subset of the specified objects on the basis of their states. The state modifier values are:

DOWN	Applies the command to only the objects that are DOWN.
NOT-UP	Applies the command to the objects that are DOWN or PENDING.
NOT-DOWN	Applies the command to the objects that are UP or PENDING.
UP	Applies the command to only the objects that are UP.

  See “DSNM Object States” on page 2-6 for the DSNM definitions for UP, DOWN, and PENDING states. The state modifiers are not used with the INFO and STATISTICS commands.

- Response modifier—controls the response information from an INQUIRE or STATUS command. The response modifier values are:

BRIEF	Returns one line of status information for each object, including the object’s DSNM state (see “DSNM Object States” on page 2-6). Also returns the object’s subsystem state, if it provides additional information. This is the default value.
DETAIL	Returns one line of status information for each object, followed by available detailed status information.
SUMMARY	Returns a one-line display showing the total number of objects that are UP, PENDING, DOWN, UNDEFINED, and IN ERROR.
SUMMARY-BYOBJECT	Returns one line of status information followed by a summary status line for each subordinate object type. Each summary status line includes the total number of objects that are UP, PENDING, DOWN, UNDEFINED, and IN ERROR.
SUMMARY-BYTYPE	Returns one line for each object type showing the total number of objects that are UP, PENDING, DOWN, UNDEFINED, and IN ERROR. The value of the *`hierarchy-modifier`* determines whether the totals include the specified objects, subordinate objects, or both.

- Highlight modifier—limits the scope of the UPDATE and INQUIRE commands to a subset of the objects, based on the objects' attributes in the DSNM object database. The highlight modifier values are:

FROM-DISPLAY	Causes the command to use the names on the NetStatus display instead of resolving them through Distributed Name Service (DNS), thus speeding name resolution. Refer to the *User's Guide to DSNM Commands* for restrictions on using this parameter.
DEFINED	Limits the scope of the command to objects defined in the DSNM object database.
UNDEFINED	Limits the scope of the command to objects not defined in the DSNM object database.
MONITORED	Limits the scope of the command to objects recorded in the DSNM object database as being monitored.
NOT-MONITORED	Limits the scope of the command to objects recorded in the DSNM object database as not being monitored.
ACCEPTABLE	Limits the scope of the command to objects that are recorded in the DSNM object database as being monitored and currently in one of their acceptable states.
UNACCEPTABLE	Limits the scope of the command to objects that are recorded in the DSNM object database as being monitored and not currently in one of their acceptable states.

Parameters

Parameters affect the action of a command and are used only with the UPDATE and STATISTICS commands. Possible UPDATE command parameters are:

ACCEPT	Changes the acceptable state of the specified objects to the states indicated.
MONITOR	Determines whether the specified objects are monitored or not when they appear on the NetStatus display.
NOMONITOR	Determines whether the specified objects are not monitored when they appear on the NetStatus display.

The STATISTICS command parameter is RESET. If you specify RESET, DSNM directs the subsystem to reset the statistical counters for the specified objects after executing the command.

Considerations

The following considerations apply to all DSNM commands except the AGGREGATE command. For more information on object specification and modifiers, refer to the *User's Guide to DSNM Commands*.

- You can use parentheses to nest object lists. Any modifiers that appear inside parentheses are limited to the object specifications within the parentheses, overriding any modifiers that apply to the entire command.
- Modifiers can appear in any order as long as there is only one of each type. The hierarchy modifier is the only exception; it can occur as part of the object specification or as part of the command as a whole, or both.
- If you specify more than one modifier of the same type, DSNM only uses the last one. You can specify a hierarchy modifier within any of the object specifications, and you can apply one to the entire command. If a hierarchy modifier is specified within an object specification, it overrides the command's hierarchy modifier.
- If you enter a command that is syntactically correct, except that it includes an incorrect or incomplete object specification, DSNM executes the command for all of the objects that can be resolved.

DSNM Object States

One purpose of DSNM is to present a uniform representation of objects and their subsystems for status displays. To this end, subsystem objects are classified into one of a small set of DSNM states. This set of states may be smaller than the possible set of subsystem states for the object. The subsystem interface process must be able to map the states of the subsystem objects to the following DSNM object states:

DOWN	The object is unavailable or needs an operator to take action to make it ready.
UP	The object in use or available for immediate use.
PENDING	The object is neither ready for use nor totally deactivated; it is in some intermediate state such as STARTING.
UNDEFINED	The object is not configured.
UNKNOWN	The state of the object cannot be determined.

Canceling Commands

If you are using NetCommand, you can cancel any command still in progress by pressing the Break key. When you press the Break key, you see the following prompt:

```
Cancel?
```

Enter Y to cancel the command; enter N to permit it to continue. Pressing the Enter key also permits the command to continue. If you enter Y, the command is immediately canceled. Any portions of the command that were already completed remain in effect.

The command server initiates a cancellation by sending a cancel buffer (where _INPUT.MOD.Z^AMOD = ZDSN^AMOD^CANCEL) to the frame. The frame then redispatches the thread with an _EV^CANCEL event. The command thread is responsible for cleaning up its environment.

Refer to the *NetStatus User's Guide* to find out how to cancel commands issued from the command line in NetStatus.

Note. You cannot cancel a command that is in progress in the NonStop NET/MASTER MS environment.

The remainder of this section provides syntax information by command, alphabetically.

The ABORT Command

The ABORT command causes DSNM to issue the subsystem-specific command(s) that stops each object. When an object is stopped, its state changes to the subsystem state that corresponds to the DSNM DOWN state. The ABORT command stops objects without waiting for any outstanding operations to be completed. This command is more emergency-oriented than the STOP command, which stops objects after completing outstanding operations.

If all objects specified in the command are stopped, there is no command response. If the command fails to stop any object, a response message lists the objects that were not stopped. If the command contains a syntax error, you receive an error message, and the objects are not stopped.

```
ABORT objectspec [, objectspec ]...
                 [, hierarchy-modifier ]
                 [, error-modifier ]
                 [, state-modifier ]
```

objectspec

is the object specification. The syntax for the object specification is provided in “Object Specification” on page 2-2.

hierarchy-modifier

determines which objects are affected by the command, based on the subsystem object hierarchy: ONLY, SUBONLY, or ALL (default).

error-modifier

determines how much information is reported when the command is correct but the objects against which the command is being applied produce errors. The error modifier does not affect the command response for errors that result when a command is entered incorrectly or when a name cannot be resolved. Possible error modifiers are ERROR-BRIEF (default), ERROR-DETAIL, and ERROR-SUPPRESS.

state-modifier

restricts the scope of the command to a subset of the specified objects, depending on their states: UP, NOT-UP, DOWN, or NOT-DOWN.

If you do not specify a value for the state modifier, DSNM applies the command to all objects that match your object specification, regardless of their states.

Considerations

Since the purpose of the ABORT command is to bring objects into the DOWN state, the command has no effect on objects already in the DOWN state.

Because subsystems generally must abort objects in a predetermined order, the ONLY hierarchy modifier is ineffective with certain object types if their subordinate objects are still up; that is, aborting certain object types forces their subordinates to be aborted also.

The ABORT command is not appropriate for all object types; refer to the *User's Guide to DSNM Commands* for details.

Example

The following command stops the Expand lines in the group ALL-EXPAND:

```
ABORT ALL-EXPAND
```

The default hierarchy modifier is ALL, but because the group consists only of lines that have no subordinate objects, there are no subordinate objects to stop.

The AGGREGATE Command

Use the AGGREGATE command to obtain the status of each object under the specified manager process or subsystem. The response message comprises information that is collated into a summary of the number of objects of each type that are in the state UP, PENDING, DOWN, UNDEFINED, or IN ERROR.

```
AGGREGATE [subsys] [\node]... [[UNDER [\node.]$manager]]
   [,[subsys] [\node]... [[UNDER [\node.]$manager]]]...
```

subsys

is the subsystem for which you want the aggregate status.

node

is the node for which you want the aggregate status.

[*node*.]$*manager*

is the name of the manager process for which you want the aggregate status (specify it only if it is applicable to the specified subsystem).

Considerations

If you specify a node each for both the name and manager process, the manager node takes precedence.

Where you do not specify a node, DSNM uses the local node.

If the subsystem requires a manager process, you must specify it.

Example

The following command returns the aggregate status of all the SNAX objects on \BERLIN:

```
AGGREGATE SNAX \BERLIN
```

A sample response to the command is:

```
SNAX LINE  \BERLIN
 1 Up, 0 Pending, 1 Down, 0 Undefined, 0 In Error
SNAX PU    \BERLIN
 1 Up, 0 Pending, 1 Down, 0 Undefined, 0 In Error
SNAX LU    \BERLIN
 5 Up, 0 Pending, 5 Down, 0 Undefined, 0 In Error
```

The INFO Command

Use the INFO command to obtain configuration information on objects.

The INFO command is not appropriate for all object types. Refer to the *User's Guide to DSNM Commands* for information on how the command is interpreted by each subsystem.

```
INFO objectspec [, objectspec ]...
                [, hierarchy-modifier ]
                [, error-modifier ]
```

objectspec

is the object specification. The syntax for the object specification is provided in "Object Specification" on page 2-2.

hierarchy-modifier

determines which objects are affected by the command, based on the subsystem object hierarchy: ONLY, SUBONLY, or ALL (default).

error-modifier

determines how much information is reported when the command is correct but the objects against which the command is being applied produce errors. The error modifier does not affect the command response for errors that result when a command is entered incorrectly or when a name cannot be resolved. Possible error modifiers are ERROR-BRIEF (default), ERROR-DETAIL, and ERROR-SUPPRESS.

Example

The following command returns the configuration information for the PATHMON-controlled terminal DPWT3 under \BERLIN.$PMD with the alias TERM-3:

```
INFO TERM-3
```

A sample response to the command is:

```
PATHWAY   TERM      DPWT3 UNDER \BERLIN.$PMD
   Autorestart: 0
   Break: off
   Diagnostic: on
   Displaypages: -1
   Echo: on
   Exclusive: off
   File: \BERLIN.$TM13B
   Initial: ENABLE-RELEASE
   Inspect: off
   Inspectfile:
   Ioprotocol: 0
   Maxinputmsgs: 0
   Printerattached: no
```

```
Printerfile:
Tclprog: \BERLIN.$SYSLOG.TRSYS.POBJ
TCP: TCP2
TMF: on
Termtype: CONVERSATIONAL
Termsubtype: 0
Trailingblanks: on
```

The INQUIRE Command

Use the INQUIRE command to obtain the current status of objects as recorded in the DSNM object database. Response time to the INQUIRE command can be faster than that of the STATUS command, but the response to the STATUS command can be more up-to-date than that of the INQUIRE command. See "Considerations" for details.

```
INQUIRE objectspec [, objectspec ]...
                   [, hierarchy-modifier ]
                   [, error-modifier ]
                   [, state-modifier ]
                   [, response-modifier ]
                   [, highlight-modifier ]
```

objectspec

is the object specification. The syntax for the object specification is provided in "Object Specification" on page 2-2.

hierarchy-modifier

determines which objects are affected by the command, based on the subsystem object hierarchy: ONLY, SUBONLY, or ALL (default).

error-modifier

determines how much information is reported when the command is correct but the objects against which the command is being applied produce errors. The error modifier does not affect the command response for errors that result when a command is entered incorrectly or when a name cannot be resolved. Possible error modifiers are ERROR-BRIEF (default), ERROR-DETAIL, and ERROR-SUPPRESS.

state-modifier

restricts the scope of the command to a subset of the specified objects, depending on their states: UP, NOT-UP, DOWN, or NOT-DOWN.

If you do not specify a value for the state modifier, DSNM applies the command to all objects that match your object specification, regardless of their states.

response-modifier

determines how much status information is returned and its format. Possible response modifiers are: BRIEF, DETAIL, SUMMARY, SUMMARY-BYOBJECT, and SUMMARY-BYTYPE.

highlight-modifier

determines the scope of the command, based on information in the DSNM object database. Possible highlight modifiers are: FROM-DISPLAY, DEFINED, UNDEFINED, MONITORED, NOT-MONITORED, ACCEPTABLE, and UNACCEPTABLE. There is no default for the highlight modifier.

FROM-DISPLAY causes the command to use the names on the NetStatus display instead of resolving them through Distributed Name Service (DNS), thus speeding name resolution. The FROM-DISPLAY parameter is valid only if you are issuing a command from the NetStatus command line and all objects specified in the command appear on the NetStatus screen. (Refer to the *NetStatus User's Guide* for more information on FROM-DISPLAY.)

Considerations

Because the INQUIRE command obtains status information from the object database, it has a faster response time than the STATUS command. This is especially noticeable in large networks. However, because the STATUS command obtains status information from the subsystems directly, it produces more up-to-date information. Take your immediate needs into account when choosing between using INQUIRE and STATUS.

Because the SUMMARY, SUMMARY-BYOBJECT, and SUMMARY-BYTYPE response modifiers return the number of objects in each state, the state modifier is ineffective with them. DSNM ignores the state modifier if it is combined with any of these response modifiers.

Because the INQUIRE command retrieves status information from the in-memory copy of the DSNM object database, it does not return the status of dynamic objects, such as PATHMON-controlled terminals (which are not added to the database), or of objects added to your network configuration after the DSNM object database was built.

Example

The following command returns status information for all the PATHMON-controlled TCPs controlled by manager process \LONDON.$PMUK that are in either the UP or PENDING state:

```
INQUIRE TCP * UNDER $PMUK, NOT-DOWN
```

Because no hierarchy modifier is specified, this command also returns the status of all the UP or PENDING terminals controlled by that TCP. The default response modifier BRIEF returns one line of information for each object. A sample response to this command is:

```
PATHWAY TCP   TCP1 UNDER \LONDON.$PMUK Up
PATHWAY TERM  UKPWT1 UNDER \LONDON.$PMUK Up
PATHWAY TERM  UKPWT4 UNDER \LONDON.$PMUK Pending
PATHWAY TERM  UKPWT5 UNDER \LONDON.$PMUK Up
PATHWAY TERM  UKPWT6 UNDER \LONDON.$PMUK Pending
PATHWAY TERM  UKPWT7 UNDER \LONDON.$PMUK Pending
PATHWAY TERM  UKPWT8 UNDER \LONDON.$PMUK Up
PATHWAY TERM  UKPWT10 UNDER \LONDON.$PMUK Up
```

The START Command

The START command causes DSNM to issue the subsystem-specific command(s) that change each object to the subsystem state that corresponds to the DSNM UP state. If all objects specified in the command are started, there is no command response. If the command fails to start any of the objects, there is a response, listing the objects that were not started. If the command line contains a syntax error, you receive an error message.

```
START objectspec [, objectspec ]...
                 [, hierarchy-modifier ]
                 [, error-modifier ]
                 [, state-modifier ]
```

objectspec

is the object specification. The syntax for the object specification is provided in “Object Specification” on page 2-2.

hierarchy-modifier

determines which objects are affected by the command, based on the subsystem object hierarchy: ONLY, SUBONLY, or ALL (default).

error-modifier

determines how much information is reported when the command is correct but the objects against which the command is being applied produce errors. The error modifier does not affect the command response for errors that result when a command is entered incorrectly or when a name cannot be resolved. Possible error modifiers are ERROR-BRIEF (default), ERROR-DETAIL, and ERROR-SUPPRESS.

state-modifier

restricts the scope of the command to a subset of the specified objects, depending on their states: UP, NOT-UP, DOWN, or NOT-DOWN.

If you do not specify a value for the state modifier, DSNM applies the command to all objects that match your object specification, regardless of their states.

Considerations

In some cases, the SUBONLY modifier has no meaning when issued with the START command. Some subsystems prevent you from requesting to start subordinate objects without starting the objects to which they are subordinate. The SNAX/XF subsystem requires that SNAX lines be started before PUs and LUs. PATHMON requires that TCPs be started before terminals.

Example

The following command starts the SNAX lines identified by the aliases BERLIN-ATM-LINE and PARIS-ATM-LINE but does not start any subordinate PUs or LUs:

```
START BERLIN-ATM-LINE PARIS-ATM-LINE, ONLY
```

The STATISTICS Command

Use the STATISTICS command to obtain operational statistics about objects. Some subsystems return statistics on only objects that are up.

The STATISTICS command is not appropriate for all object types; refer to the *User's Guide to DSNM Commands* for details on this restriction.

```
STATISTICS objectspec [, objectspec ]...
                      [, hierarchy-modifier ]
                      [, error-modifier ]
                      [, RESET ]
```

objectspec

is the object specification. The syntax for the object specification is provided in "Object Specification" on page 2-2.

hierarchy-modifier

determines which objects are affected by the command, based on the subsystem object hierarchy: ONLY, SUBONLY, or ALL (default).

error-modifier

determines how much information is reported when the command is correct but the objects against which the command is being applied produce errors. The error modifier does not affect the command response for errors that result when a command is entered incorrectly or when a name cannot be resolved. Possible error modifiers are ERROR-BRIEF (default), ERROR-DETAIL, and ERROR-SUPPRESS.

RESET

If you specify RESET, DSNM directs the subsystem to reset the statistical counters for the specified objects after executing the command. Not all counters are necessarily reset; the subsystems determine which counters are reset for different types of objects.

Example

The following command returns the operational statistics for the Expand line \PARIS.$LHD (local node is \LONDON):

```
STATISTICS \PARIS.$LHD
```

A sample response to the command is:

```
EXPAND             LINE              \PARIS.$LHD
   Current Timestamp:  21 Feb 1992, 16:07:52.044
   Last Resetstats Time:  21 Feb 1992, 16:03:36.351
   I Frames Sent:  17
   I Frames Rcvd:  8
   S Frames Sent:  25
   S Frames Rcvd:  39
   U Frames Sent:  0
   U Frames Rcvd:  0
   L2 I Frames Sent:  17
   L2 I Frames Rcvd:  8
   L2 I Frames Sent P:  0
   L2 I Frames Rcvd P:  0
   L2 RR Frames Sent:  25
   L2 RR Frames Rcvd:  39
   L2 RNR Frames Sent:  0
   L2 RNR Frames Rcvd:  0
   L2 REJ Frames Sent:  0
   L2 REJ Frames Rcvd:  0
   L2 SABM Frames Sent:  0
   L2 SABM Frames Rcvd:  0
   L2 DISC Frames Sent:  0
   L2 DISC Frames Rcvd:  0
   L2 CMDR Frames Sent:  0
   L2 CMDR Frames Rcvd:  0
   L2 UA Frames Sent:  0
   L2 UA Frames Rcvd:  0
    L2 DM Frames Sent:  0
   L2 DM Frames Rcvd:  0
   L2 SREJ Frames Sent:  0
   L2 SREJ Frames Rcvd:  0
   L2 FCS Errors:  0
   L2 Timeouts:  0
   L2 Address Errors:  0
   L2 Length Errors:  0
   L2 Receive Aborted:  0
   L2 Received Buffers:  0
   Driver Frames Total:  0
   Driver Frames Error:  0
   Driver NO Buffer:  0
   Driver BCC Errors:  0
   Driver Line Quality:  0
   Driver Receive Overrun:  0
   Driver Modem Errors:  0
   CTS State:  On
   DCD State:  On
   DSR State:  On
   Primary PID:  (2,44)
   Backup PID:  (3,25)
```

The STATUS Command

Use the STATUS command to obtain the current subsystem status of objects, as obtained from the subsystem. The STATUS command generates more up-to-date information than the INQUIRE command, but the response time can be slower.

```
STATUS objectspec [, objectspec ]...
                  [, hierarchy-modifier ]
                  [, error-modifier ]
                  [, state-modifier ]
                  [, response-modifier ]
```

objectspec

is the object specification. The syntax for the object specification is provided in “Object Specification” on page 2-2.

hierarchy-modifier

determines which objects are affected by the command, based on the subsystem object hierarchy: ONLY, SUBONLY, or ALL (default).

error-modifier

determines how much information is reported when the command is correct but the objects against which the command is being applied produce errors. The error modifier does not affect the command response for errors that result when a command is entered incorrectly or when a name cannot be resolved. Possible error modifiers are ERROR-BRIEF (default), ERROR-DETAIL, and ERROR-SUPPRESS.

state-modifier

restricts the scope of the command to a subset of the specified objects, depending on their states: UP, NOT-UP, DOWN, or NOT-DOWN. If you do not specify a value for the state modifier, DSNM applies the command to all objects that match your object specification, regardless of their states.

response-modifier

determines how much status information is returned and it s format. Possible response modifiers are: BRIEF, DETAIL, SUMMARY, SUMMARY-BYOBJECT, and SUMMARY-BYTYPE.

Considerations

Because the STATUS command obtains status information from the subsystems directly, it produces more up-to-date information than the INQUIRE command. However, because the INQUIRE command obtains status information from the object database, it has a faster response time than the STATUS command. This is especially noticeable in large networks.

The state modifier is ignored if the response modifier has the value of SUMMARY, SUMMARY-BYTYPE, or SUMMARY-BYOBJECT, because these values return totals by subsystem state.

If the SUBONLY and SUMMARY modifiers are combined for object types on the lowest level, counter values are 0. (Subordinate objects do not exist; therefore, the summary of their counters is 0.)

DETAIL is not a valid response modifier for Tandem data communications subsystems other than AM3270, Expand, SNAX, SNAX/CDF, TR3271, and X25AM.

The STATUS command is not appropriate for all object types; refer to the *User's Guide to DSNM Commands* for details on these restrictions.

Example

The following command returns the total number of objects in each state for TCP1 and its subordinate terminals:

```
STATUS TCP1 UNDER $PMUK, SUMMARY
```

A sample response to the command is:

```
7 Up, 1 Pending, 2 Down, 0 Undefined, 0 In Error
```

The following command returns a summary line of status information for each object type on each node included in the members of the group ALL-PATHWAY:

```
STATUS ALL-PATHWAY, SUMMARY-BYTYPE
```

A sample response to the command is:

```
PATHWAY SERVER   \LONDON
   2 Up,  0 Pending, 0 Down, 0 Undefined, 0 In Error
PATHWAY TCP      \LONDON
   1 Up,  0 Pending, 0 Down, 0 Undefined, 0 In Error
PATHWAY TERM     \LONDON
  10 Up,  0 Pending, 0 Down, 0 Undefined, 0 In Error
PATHWAY SERVER   \BERLIN
   1 Up,  1 Pending, 0 Down, 0 Undefined, 0 In Error
PATHWAY TCP      \BERLIN
   0 Up,  1 Pending, 0 Down, 0 Undefined, 0 In Error
PATHWAY TERM     \BERLIN
   0 Up, 10 Pending, 0 Down, 0 Undefined, 0 In Error
```

The STOP Command

Use the STOP command to stop objects. The command causes DSNM to issue the appropriate subsystem commands to stop each object after all current and outstanding operations are complete.

If all objects specified in the command are successfully stopped, there is no command response. If the command fails to stop any object, there is a response, listing the objects that were not stopped. If the command contains a syntax error, you receive an error message, and the objects are not stopped.

If you wish to stop objects immediately, regardless of outstanding operations, use the ABORT command.

```
STOP objectspec  [, objectspec ]...
                 [, hierarchy-modifier ]
                 [, error-modifier ]
                 [, state-modifier ]
```

objectspec

is the object specification. The syntax for the object specification is provided in “Object Specification” on page 2-2.

hierarchy-modifier

determines which objects are affected by the command, based on the subsystem object hierarchy: ONLY, SUBONLY, or ALL (default).

error-modifier

determines how much information is reported when the command is correct but the objects against which the command is being applied produce errors. The error modifier does not affect the command response for errors that result when a command is entered incorrectly or when a name cannot be resolved. Possible error modifiers are ERROR-BRIEF (default), ERROR-DETAIL, and ERROR-SUPPRESS.

state-modifier

restricts the scope of the command to a subset of the specified objects, depending on their states: UP, NOT-UP, DOWN, or NOT-DOWN. If you do not specify a value for the state modifier, DSNM applies the command to all objects that match your object specification, regardless of their states.

Considerations

An object engaged in a lengthy operation can take a long time to stop. Therefore, the subsystem commands issued by DSNM to stop the objects, and thus the DSNM STOP command itself, can be complete before the objects actually stop. Consequently, some of the objects can still be stopping for some time after the STOP command is issued,

during which time the objects can be reported by DSNM as in either the DOWN or the PENDING state.

Because each subsystem must stop its objects in a predetermined order, the ONLY hierarchy modifier is ineffective with certain object types if their subordinate objects are still up; that is, stopping certain object types forces their subordinates to also be stopped.

The STOP command is not appropriate for all object types; refer to the *User's Guide to DSNM Commands* for details on these restrictions.

Example

The following command stops the PUs and LUs that are subordinate to the SNAX line \WYJ.$STLR. It does not stop the SNAX line itself. No error information is returned if a subsystem error occurs while the command is being executed.

```
STOP SNAX LINE \WYJ.$STLR, SUB-ONLY, ERROR-SUPPRESS
```

The UPDATE Command

Use the UPDATE command to modify object attributes in the DSNM object database that control:

- Whether an object is monitored when it appears on the NetStatus display
- What states cause an object to be highlighted on the NetStatus display

The UPDATE command returns a response for only those objects that are not successfully updated.

The UPDATE command line must include a MONITOR, NOMONITOR, or ACCEPT parameter.

```
UPDATE objectspec [, objectspec ]...
                  [, hierarchy-modifier ]
                  [, error-modifier ]
                  [, state-modifier ]
                  [, highlight-modifier ]
                  [, MONITOR | NOMONITOR ]
                  [, ACCEPT [ UP ] [ DOWN ] [ PENDING ] ]
```

objectspec

is the object specification. The syntax for the object specification is provided in “Object Specification” on page 2-2.

hierarchy-modifier

determines which objects are affected by the command, based on the subsystem object hierarchy: ONLY, SUBONLY, or ALL (default).

error-modifier

determines how much information is reported when the command is correct but the objects against which the command is being applied produce errors. The error modifier does not affect the command response for errors that result when a command is entered incorrectly or when a name cannot be resolved. Possible error modifiers are ERROR-BRIEF (default), ERROR-DETAIL, and ERROR-SUPPRESS.

state-modifier

restricts the scope of the command to a subset of the specified objects, depending on their states: UP, NOT-UP, DOWN, or NOT-DOWN. If you do not specify a value for the state modifier, DSNM applies the command to all objects that match your object specification, regardless of their states.

highlight-modifier

determines the scope of the command, based on information in the DSNM object database. Possible highlight modifiers are: FROM-DISPLAY, DEFINED,

UNDEFINED, MONITORED, NOT-MONITORED, ACCEPTABLE, and UNACCEPTABLE. There is no default for the highlight modifier.

FROM-DISPLAY causes the command to use the names on the NetStatus display instead of resolving them through Distributed Name Service (DNS), thus speeding name resolution. The FROM-DISPLAY parameter is valid only if you are issuing the UPDATE command from the NetStatus command line and all the objects specified in the command appear on the NetStatus screen. (Refer to the *NetStatus User's Guide* for more information.)

Note. Objects not defined in the object database are excluded by all values of the highlight modifier except UNDEFINED.

`MONITOR | NOMONITOR`

determines whether the specified objects are to be monitored or not when they appear on the NetStatus display.

Note. The MONITOR and NOMONITOR parameters are similar in name to two values of the highlight modifier; be careful to distinguish them:

- MONITOR and NOMONITOR are parameters that can be used with the UPDATE command to switch monitoring on and off.
- MONITORED and NOT-MONITORED are values of the highlight modifier that limit the scope of the command either to objects currently being monitored or to objects not currently being monitored.

`ACCEPT [UP] [DOWN] [PENDING]`

changes the acceptable state of the specified objects to the states indicated. If no states are indicated, the current state becomes the acceptable state. Objects not in an acceptable state are highlighted when they appear on the NetStatus display.

Considerations

If you specify ACCEPT with no parameters, DSNM replaces the values in the acceptable states field of the object entry with the current state of the object. If you specify more than one subsystem object in the UPDATE command line, DSNM replaces the acceptable states for each object with that object's current state. If you specify ACCEPT followed by any combination of the keywords UP, DOWN, and PENDING, DSNM replaces the current acceptable states for each object with the states that you specify.

The UPDATE command is not supported in NonStop NET/MASTER MS.

Example

The following command replaces the current values of the acceptable states fields for the SNAX PU \BERLIN.$SATM.#ATMCC with the current state of the PU:

```
UPDATE PU \BERLIN.$SATM.#ATMCC, ONLY, ACCEPT
```

Because the command line includes the ONLY modifier, it does not update the object entries for any subordinate LUs.

3 I Process Development Process

Scope of This Section

This section introduces the conceptual model upon which the I process program's frame and command thread interactions and supporting library services are based. Also provided are a detailed development model and associated rules for using the I process development software correctly and effectively.

The purpose of this section is to:

- Define central concepts of the I process program structure.
- Describe the environment and services provided for I process development, focusing on the command thread and how it interacts with the program frame and library services.
- Outline how to integrate your code with the frame code to produce a working I process.

Complete syntax and parameter descriptions for all procedures, literals, defines, and structure templates discussed in this section are provided in Appendix A, "DSNM Library Services." ZDSN error codes are described in Appendix B, "DSNM Error Codes." The SPI DDL constants and structure definitions that users must know about are listed in Appendix C, "Data Definition Language (DDL)-Defined DSNM SPI Components."

Note. Appendix D, "Sample I Process Program Code," contains an example of the user-written portion of an I process program, illustrating the model and associated library services described in this section. You may find it helpful to refer to this sample code when reading this section.

Function of the I Process

You add a Tandem or customer-written subsystem to the existing base of DSNM-managed subsystems by providing a DSNM subsystem interface process, commonly referred to as an "I process." An I process converts DSNM commands affecting subsystem objects into a sequence of subsystem commands, sends them to the targeted subsystem's control interface for processing, and converts the result into a standard DSNM form.

An I process is a member of a DSNM server class. As highlighted in Figure 3-1, it:

- Receives commands from a DSNM requester (usually the command server).
- Converts DSNM commands into syntactically correct subsystem commands.
- Presents commands to the subsystem CI for execution.
- Converts the result into a standard DSNM form.
- Returns responses to the requester.

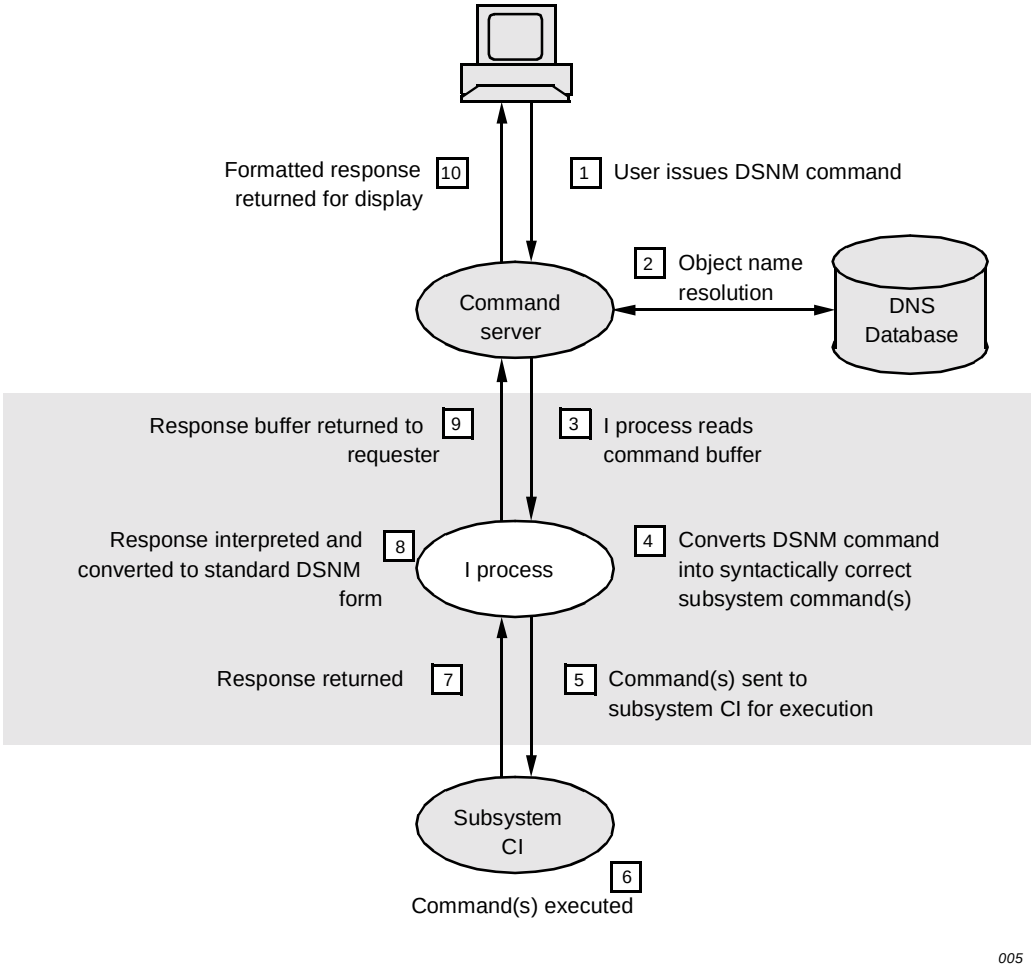
You can think of an I process as being a translator from the DSNM language to the language supported by the subsystem CI.

Note. This release of the DSNM subsystem interface development software addresses server-type CIs only, which must be started outside the I process.

DSNM commands can be categorized by function as follows:

- Information retrieval commands—return various types of information about objects. These are the AGGREGATE, INFO, STATISTICS, and STATUS commands.
- Control commands—change the state of objects. These are the START, STOP, and ABORT commands.
- Monitoring commands—report information on the status of subsystem objects as currently stored in the DSNM object database, and enable you to control certain aspects of how objects are monitored. These are the INQUIRE and UPDATE commands.

Figure 3-1. Function of the I Process



I Process Program Structure Concepts

The following concepts are central to the I process program structure:

Frame

A frame is a set of compiled procedures supplied by Tandem into which user-written subsystem-specific code is bound to produce an I process that conforms to DSNM protocols. The frame carries out the following major functions:

- Initialization and configuration
- Thread management
- Communication with the DSNM requester (usually the command server)
- Communication with the subsystem CI (any gateway to the subsystem for control purposes)

△ **Caution.** All $RECEIVE operations are done by the frame on behalf of the I process; the frame does this by calling internal DSNM library procedures. Do not perform any NonStop Kernel file operations that affect $RECEIVE, such as FILE_OPEN_ or FILE_GETRECEIVEINFO_. Processes may fail or behave unpredictably if you attempt to open $RECEIVE.

Thread

A thread is an independent instruction stream capable of being interleaved in execution with other instruction streams (under the control of the frame). In this manual, an executing instance of a command thread is called simply “a thread,” and the procedures that compose it are called the “thread procedures.”

You use I process-development library services to write TAL procedures executed by the frame as an independent thread, called the “command thread.” These procedures collectively perform the following major functions:

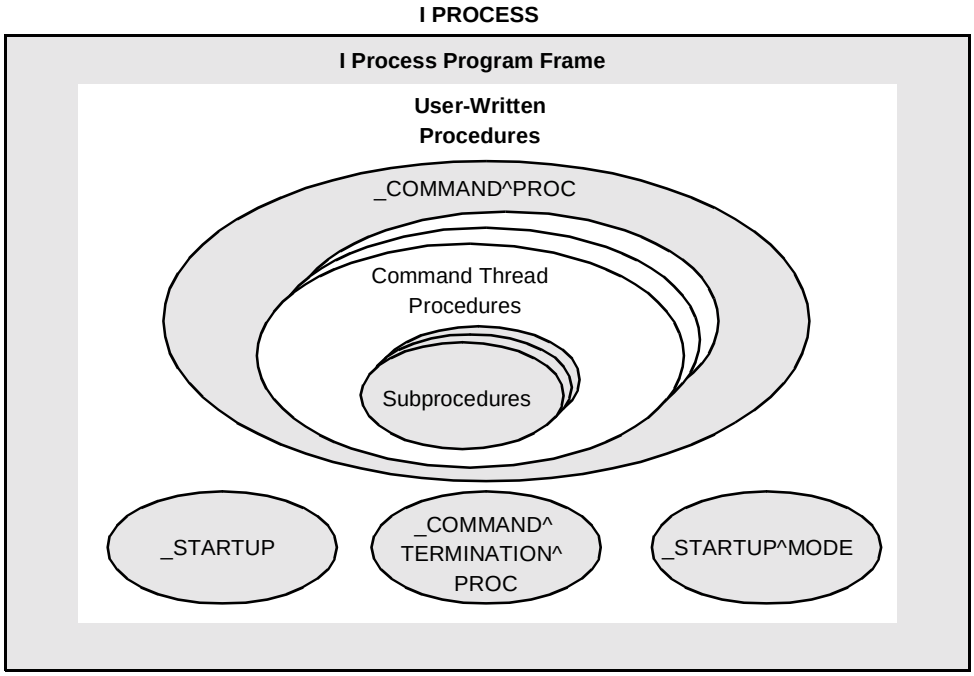
- Translation of DSNM commands into subsystem-specific commands
- Construction of command buffers to be sent to the subsystem CI
- Interpretation of response buffers returned by the subsystem CI

The frame handles the bulk of the complexity of the requester-to-I process and process-to-CI interactions. Library services support frame and user-written operations such as memory management and list processing.

Figure 3-2 illustrates the relationship between the I process program frame supplied by Tandem and the user-written procedures _COMMAND^PROC, _STARTUP, _STARTUP^MODE, and _COMMAND^TERMINATION^PROC.

See Table 3-1 for an overview of the defines, procedures, structure templates, and user-written procedure identifiers referenced in this illustration and in code examples throughout this section.

Figure 3-2. Relationship Between the Frame and User-Written Procedures



Dispatch

Dispatch means invoking a thread for execution.

A thread procedure periodically suspends execution until something occurs (for example, the completion of an I/O operation) by returning to the frame with a RETURN procedure and waiting for the frame to generate a particular event, at which point the frame dispatches the current thread procedure. (See the “Event” discussion next in this subsection.) As the thread executes, it can alter the current thread procedure to be called by the frame at the next thread dispatch.

When execution proceeds after an event, the current thread procedure is reentered from the beginning. The thread procedure determines its current state from the event that occurred and information stored in its context area (see the “Command Context” discussion later in this subsection).

Library routines help support state maintenance and restoration. Local variables are not preserved and must be reinitialized on continuation after any return to the frame.

Event

An event is an occurrence that initiates a thread dispatch. Whenever a thread is dispatched, the event that caused the dispatch is communicated to the thread.

Command Context

A command context is a collection of data in memory reserved for a particular thread's exclusive use during its execution. When the frame receives a command, it creates an instance of the command thread and dynamically allocates memory for a command context area that is preserved for the life of the thread. Thread procedures must use memory within their context area.

Formatted Object

A formatted object is the data structure, defined by the ZDSN^DDL^FOBJECT^DEF DDL structure, that defines a subsystem object to DSNM. The fields within the structure contain information about all the relevant attributes of a subsystem object. Each object that a command has to act on is defined as a filled-in formatted object structure.

List

A list is a double-ended queue structure that consists of a list declaration and list members:

- The list declaration is a small data structure that holds control information for use by the I process memory management services. Its size and structure are fixed.
- A list member is a block of memory, the size and description of which are determined by the thread. Memory is allocated dynamically as members are added to a list, and deallocated as members are removed from a list. List members can be of any size.

Two predefined list structures are available in the thread's command context space:

- The input object list is the list of objects (each one of which is represented by a formatted object structure) upon which a DSNM operation is performed.
- The output object list is produced by the command thread when processing the command; it is initially empty.

Library functions support creation of additional lists for intermediate data storage.

CI

A CI is a control interface: any gateway to the subsystem that provides control. Within the I process model, a CI is conceptually the name of a control interface, analogous to a NonStop Kernel process name. Like a NonStop Kernel process, a CI can be opened for communication. An open CI is referred to by a *cid*, which is the functional equivalent of a NonStop Kernel file number for an open NonStop Kernel process. _SEND^CI is an operation that provides for communication with an open CI.

General Command Processing Scheme

As an I process developer, you write a set of procedures that, when executed, form a command thread that carries out a DSNM command. The general command processing scheme is listed next. (See Figure 3-3 on page 3-8 for an illustrated example.)

1. When the frame receives a command, it extracts the following command components and places them in the command context space it allocates to each thread when it is created:
   - The action to be performed.
   - The command modifiers.
   - A list of objects on which the operation is to be performed (the input object list). The input object list is made up of a header and a linked list. Each list member is a formatted object structure, defining one object to which the command is applied.
2. The frame allocates the command context space, creates an instance of the command thread, and dispatches the command thread.
3. The thread's overall task is to apply the command to each object in the input object list by carrying out the following steps:
   a. In its simplest form, the thread creates a subsystem command equivalent to the DSNM command. If the subsystem does not support the operation, the thread may:
      - Perform the operation by means of a combination of subsystem commands.
      - Simulate the operation. (If the subsystem doesn’t support a particular operation, the I process itself might be coded to support it. For example, the I process might keep its own statistics on subsystem objects for which a STATISTICS operation is not supported.)
      - Treat the operation as a no-operation (but still produce the output required by the command).
      - Reject the operation with an error.

   b. The thread selects an object off the input list and sends it to an appropriate subsystem CI.

   c. The thread returns to the frame to await completion of the CI communication.

   d. When a response is received from the CI, the frame redispatches the thread.

   e. The thread interprets the response and creates an appropriate response for the command (see Section 4, “DSNM Command Requirements”).

f. The thread places the response (possibly empty, depending on the command requirements) into a predefined output object list (defined as part of the command context space).

Steps b through f are repeated until the input list is exhausted and the output list represents a complete response to the original DSNM command. During the process, the thread can define its own lists and add members to it for intermediate results.

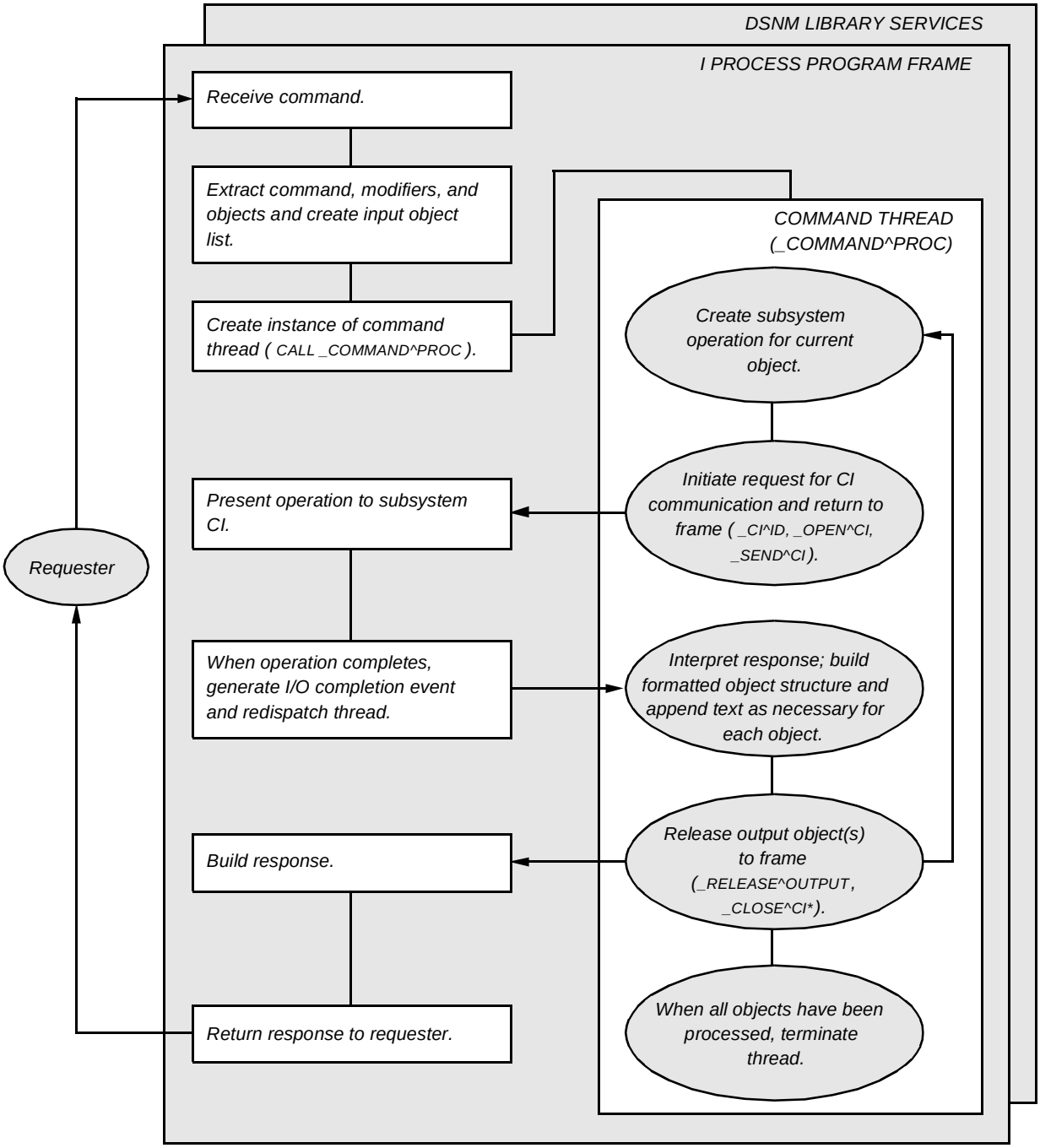
Note. Part of the command context space is a user-defined area where the thread can define and manipulate intermediate lists.

---

4. When the command has been completely processed, the thread must free all the user-allocated lists and then stop by returning an appropriate return code to the frame.

5. The frame then formats the response and returns it to the DSNM requester. The frame is responsible for freeing the input and output object lists.

Figure 3-3. Frame/Command Thread Interaction: Processing a DSNM Command



** Depending on the subsystem, a CI may be closed inside or outside the object processing loop.*

007

The Command Thread Source Environment

The source environment in which the command thread is written consists of source definitions (DDL, literals, and defines), global definitions, and external declarations. You must include the following ?SOURCE statements in your program:

```
?SOURCE KDSNDEFS (IPROCESS^DEFINITIONS)
?SOURCE KDSNDEFS (IPROCESS^GLOBALS)
?SOURCE KDSNDEFS (IPROCESS^EXTDECS)
```

Your program source file should be arranged as follows:

```
? < User compiler directives >

?SOURCE KDSNDEFS (IPROCESS^DEFINITIONS)

  < User-defined I-process globals >
```

Note. Because they are shared by all currently active threads, global definitions must be read-only after initialization.

The following templates should be defined:

```
BLOCK PRIVATE;
STRUCT in^lm^def (*);
  BEGIN                        ! Input list member definition
    _INPUT^LM^HEADER;

    < User-defined input list member fields for work space,
      if any >

  END;

STRUCT out^lm^def (*);
  BEGIN                        ! Output list member definition
    _OUTPUT^LM^HEADER;

    < User-defined output list member fields for work space,
      if any >

  END;

STRUCT cx^def (*);
  BEGIN                        ! Command context definition
    _COMMAND^CONTEXT^HEADER;
    INT .EXT inobj (in^lm^def);
    INT .EXT outobj (out^lm^def);

    < User-defined context fields >

  END;

INT .EXT ci^config (_CI^DEF);
INT .EXT ss^config (_SUBSYS^DEF);
```

```
STRING .ciname[0:ZDSN^MAX^CICLASS-1] := [“xxxxxxx ”];
STRING .ssname[0:ZDSN^MAX^SUBSYS-1] := [“xxxxxxx ”];


END BLOCK;

?SOURCE KDSNDEFS (IPROCESS^GLOBALS)
?NOLIST, SOURCE EXTDECS0 ( ... )
?LIST
?SOURCE KDSNDEFS (IPROCESS^EXTDECS)

_THREAD^PROC(MYPROC1); FORWARD;
_THREAD^PROC(MYPROC2); FORWARD;

INT PROC _STARTUP (cx^length, in^lm^length) EXTENSIBLE;
INT .cx^length, .in^lm^length;

 BEGIN
  cx^length := $LEN(cx^def);
  in^lm^length := $LEN(in^lm^def);

    IF _ISNULL (@ci^config := _ADD^CI (ciname)) THEN
      RETURN ZDSN^ERR^INTERNAL^ERR;
    IF _ISNULL (@ss^config := _ADD^SUBSYS (ssname)) THEN
      RETURN ZDSN^ERR^INTERNAL^ERR;
  RETURN ZDSN^ERR^NOERR;
 END;

INT PROC _STARTUP^MODE (component, testmode,
                        accept^startup^component,
                        subject)
                        EXTENSIBLE;

STRING .EXT component;     -- ZDSN^DDL^COMPONENT^DEF,
INT .EXT testmode;
INT .EXT accept^startup^component;
STRING .EXT subject;

BEGIN
    < move subsystem name to COMPONENT >

  testmode := _COMPILED^IN^TESTMODE;
  accept^startup^component := 1;
  RETURN ZDSN^ERR^NOERR;
END;

_THREAD^PROC(_COMMAND^PROC);
  BEGIN
    INT .EXT cx (cx^def) = _THREAD^CONTEXT^ADDRESS;
    ...
_END^THREAD^PROC;

    < other command thread procedures >
```

```
_THREAD^TERMINATION^PROC (_COMMAND^TERMINATION^PROC);
   BEGIN
    < clean up thread environment >
  _END^THREAD^TERMINATION^PROC;
```

ASSIGN Statements Required for Compilation

For compilation, you must assign the following subvolumes to be searched:

- $SYSTEM.SYSTEM—or the subvolume on your system that contains EXTDECS0.
- The volume(s) and subvolume(s) on your system that contain:

  ZDSNDEFS
  KDSNDEFS

  ZDSNLIB
  KDSNLIB

  ZSPITAL
  ZDSNTAL

- The volume and subvolume on your system that contains definitions for your subsystem.

For example:

```
ASSIGN SSV1, $SYSTEM.SYSTEM
ASSIGN SSV2, $DSNM.IDEVLIB
ASSIGN SSV3, $DSNM.IDEVDDL
```

User-Written Procedures

As illustrated in Figure 3-2 on page 3-4, user-written I process procedures consist of:

- Two startup procedures: _STARTUP^MODE and _STARTUP.
- The initial command thread procedure (_COMMAND^PROC), and other command thread procedures and utility procedures as necessary.
- An optional thread termination procedure: _COMMAND^TERMINATION^PROC.

The _STARTUP^MODE Procedure

The frame calls_STARTUP^MODE when it begins its startup processing. _STARTUP^MODE performs the following tasks:

- Retrieves the component name of the subsystem(s) being handled by the I process.
- Determines whether the I process is running in test mode.
- Determines whether to use the COMPONENT process parameter value if one appears in the startup message.

The format of the _STARTUP^MODE procedure is as follows:

```
INT PROC _STARTUP^MODE ( component
                        ,testmode
                        ,accept-startup-component
                        ,subject)
                       EXTENSIBLE;
```

component

is usually the name of the subsystem the I process handles. For I processes that handle more than one subsystem, the component name is an arbitrary name chosen by the developer of the process. For example, the SCP I process supplied by Tandem handles multiple communications subsystems: COMM is its component name. The component name is used for configuration parameter retrieval searches.

testmode

passes a value indicating whether the I process is running in test mode (which affects startup parameter processing; see Section 5, “DSNM Process Startup Functions”).

accept-startup-component

indicates whether a process COMPONENT parameter value in the startup message should (nonzero) or should not (zero, the default) override the *component* value.

subject

identifies the I process name, up to ZDSN-MAX-COMPONENT characters (36), terminated by a space or null. It is included as the subject value in all EMS messages generated by the I process.

_STARTUP^MODE is discussed further in Section 5, “DSNM Process Startup Functions.”

The _STARTUP Procedure

The frame must supply the lengths of the command context space and the input list member structures for which it allocates memory each time it creates an instance of a thread. _STARTUP declares an initialization procedure that is called by the frame to provide this information before it creates the first instance of the command thread.

_STARTUP also retrieves subsystem and CI configuration parameters from the DSNMCONF file and places them into predefined structures for use by the frame. (See “Command Context Space” on page 3-15 for a definition of the command context space.)

The format of the _STARTUP procedure is as follows:

```
INT PROC _STARTUP ( context-length,input-lm-length )
                  EXTENSIBLE;
```

context-length

is the length, in bytes, of the user-defined command context structure.

input-lm-length

is the length, in bytes, of the user-defined input list member structure.

If no values are provided, the frame allocates only the space required for its own use; no space is made available for user data.

The following procedures must be called in your _STARTUP procedure:

_ADD^SUBSYS

fills in a predefined structure with subsystem configuration parameters for the subsystem(s) the I process handles. The frame uses this information when it gets a command for that subsystem.

_ADD^CI

fills in a predefined structure with CI configuration parameters for the CI class with which your I process communicates.

The _STARTUP procedure is described in more detail in Section 5, “DSNM Process Startup Functions.”

Declaring Thread Procedures: _THREAD^PROC and _END^THREAD^PROC

Any procedure that might be dispatched as part of a thread must be declared with _THREAD^PROC and _END^THREAD^PROC:

```
_THREAD^PROC ( procname );
  BEGIN
   < procedure body >
  _END^THREAD^PROC;
```

_END^THREAD^PROC issues RETURN _RC^WAIT, which returns the thread to the frame until the occurrence of the next event causes it to be redispatched. (See "Suspending and Dispatching Thread Procedures" later in this section for more information.)

Use _THREAD^PROC in the following constructions:

```
_THREAD^PROC ( procname ); EXTERNAL;
```

```
_THREAD^PROC ( procname ); FORWARD;
```

The Initial Command Thread Procedure: _COMMAND^PROC

The first time it dispatches an instance of the thread, the frame invokes the command thread as _COMMAND^PROC. You must declare the initial thread procedure with _THREAD^PROC and _END^THREAD^PROC:

```
_THREAD^PROC (_COMMAND^PROC);
  BEGIN
    < procedure body <
  _END^THREAD^PROC;
```

The Thread Termination Procedure: _COMMAND^TERMINATION^PROC

When a thread terminates, either because of a fatal error or because it has successfully completed the processing of a command, the thread library looks for a user-written procedure named _COMMAND^TERMINATION^PROC, which may be used for cleaning up the thread's environment. You must declare this procedure with _THREAD^TERMINATION^PROC and _END^THREAD^TERMINATION^PROC:

```
_THREAD^TERMINATION^PROC ( _COMMAND^TERMINATION^PROC );
  BEGIN
   < procedure body >
  _END^THREAD^TERMINATION^PROC;
```

Use _THREAD^TERMINATION^PROC in the following construction:

```
_THREAD^TERMINATION^PROC (_COMMAND^TERMINATION^PROC);
  BEGIN
     .
     .
   < procedure body >
     .
     ! for example, free lists, close the open CI(s) and
     ! return, leaving the input and output lists for the
     ! frame
     CALL _DEALLOCATE^LIST (...);
     CALL _CLOSE^CI (...);
     .
  _END^THREAD^TERMINATION^PROC;
```

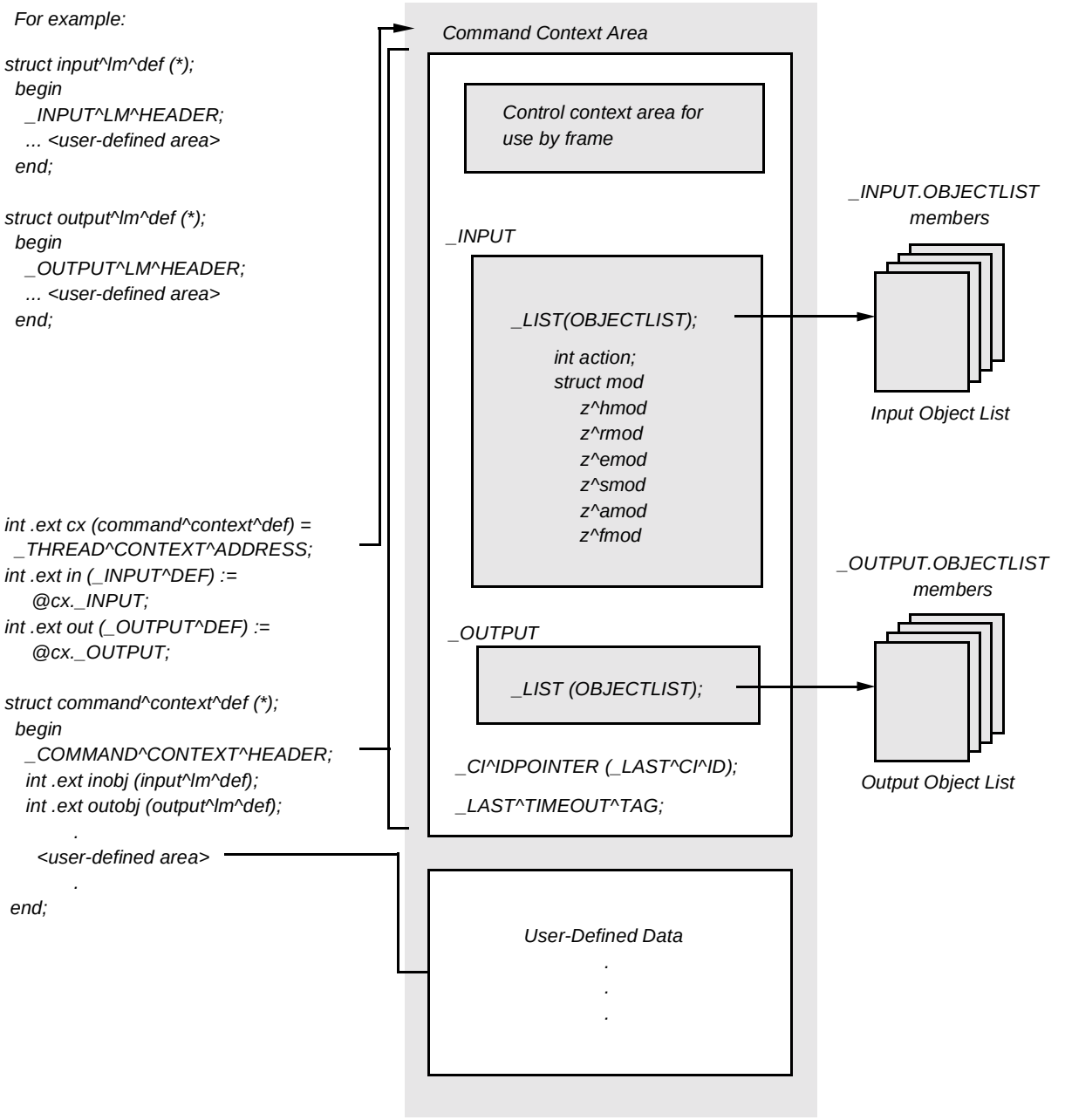
Command Context Space

When the frame receives a command, it allocates memory for a command context space and creates an instance of the command thread. You define the command context space in your globals area to include the following:

- An input area, where the frame places the following command components for access by the command thread: the command's action, modifiers, parameter list, and the input list of objects on which the command operation is performed.
- An output area, where the frame predefines the output list. The thread will place the objects with their associated states and/or text in response to the operation having been performed in the output list.
- A user area, customized by the thread.
- A control context area reserved for use by the frame for state variable maintenance and multithreading. The frame saves such variables as the current thread state, the current thread procedure, and the event(s) that caused the current dispatch in the control context area.

You define the input and output list members and the user area by specifying a structure template for each. The first part of each structure is reserved for use by the frame.

Figure 3-4. Command Context Area



Accessing the Command Context Space

_THREAD^CONTEXT^ADDRESS is an INT(32) global variable in which the frame places the extended address of the command context space before each thread dispatch.

The thread can access the command context space with a data definition similar to the following example:

```
INT.EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
```

where

`command^context^def`

is a user-defined structure that describes the command context space. The definition of `command^context^def` is provided in the next subsection, "Defining the Command Context Space."

Defining the Command Context Space

The command context space contains both frame-defined areas and a user-defined area. Your _STARTUP procedure provides the frame with the length of the command context area.

Generating the Frame-Defined Variables: _COMMAND^CONTEXT^HEADER

_COMMAND^CONTEXT^HEADER is a define that is required as part of the command context structure definition. It declares the frame-defined input, output, and control context areas.

Defining the User Area

You define the rest of the command context space according to your needs.

The following example declares a command context structure containing three user-defined lists, a CIID structure (see "CI Communications," later in this section), and several work variables:

```
STRUCT COMMAND^CONTEXT^DEF (*);
 BEGIN
  _COMMAND^CONTEXT^HEADER;
  _LIST (objlist1);
  _LIST (objlist2);
  _LIST (objlist3);
  _CI^ID (subsys^mgr);
  INT work1[0:9];
  STRING work2;
  INT(32) work3;
    .
    .
 END;
```

The Input Area: _INPUT

The input area is the portion of the command context space in which the frame places the command components.

_INPUT is the name assigned to the _INPUT^DEF structure generated by _COMMAND^CONTEXT^HEADER. This structure contains the _LIST declarations for the input object list (OBJECTLIST), the action field, and a structure containing the command modifiers.

```
STRUCT _INPUT^DEF (*);
  BEGIN
   _LIST (OBJECTLIST);
   INT action;
   STRUCT mod (zdsn^mod^def);
END;
```

_INPUT.ACTION is one of the following:

ZDSN^ACTION^ABORT
ZDSN^ACTION^AGGREGATE
ZDSN^ACTION^INFO
ZDSN^ACTION^STATUS
ZDSN^ACTION^START
ZDSN^ACTION^STATISTICS
ZDSN^ACTION^STOP

_INPUT.MOD is zero or more of the following:

Hierarchy modifier (.Z-HMOD):

ZDSN^HMOD^ALL
ZDSN^HMOD^ONLY
ZDSN^HMOD^SUBONLY

Error modifier (.Z-EMOD):

ZDSN^EMOD^BRIEF
ZDSN^EMOD^DETAIL
ZDSN^EMOD^SUPPRESS

Select state modifier (.Z-SMOD):

ZDSN^SMOD^UP | ZDSN^SMOD^GREEN
ZDSN^SMOD^NOT^UP | ZDSN^SMOD^NOT^GREEN
ZDSN^SMOD^DOWN | ZDSN^SMOD^RED
ZDSN^SMOD^NOT^DOWN | ZDSN^SMOD^NOT^RED

Response modifier (.Z-RMOD):

ZDSN^RMOD^BRIEF
ZDSN^RMOD^DETAIL

Note. SUMMARY response modifiers are handled entirely by the I process frame and are never seen by the command thread itself.

Action modifier (.Z-AMOD): ZDSN^AMOD^RESET

Flow modifier (.Z-FMOD)

Actions and modifiers are described in detail in Section 4, "DSNM Command Requirements."

Accessing the Input Area

Use data definitions similar to the following to access the input area:

```
INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT in (_INPUT^DEF) := @cx._INPUT;
```

The Output Area: _OUTPUT

The output area is the portion of the command context space in which the frame declares the output object list that will be generated as a result of processing a command.

_OUTPUT is the name assigned to the _OUTPUT^DEF structure generated by _COMMAND^CONTEXT^HEADER. This structure contains the _LIST declaration for the output object list (OBJECTLIST).

```
STRUCT _OUTPUT^DEF (*);
  BEGIN
  _LIST (OBJECTLIST);
  END;
```

Accessing the Output Area

Use data definitions similar to the following to access the output area:

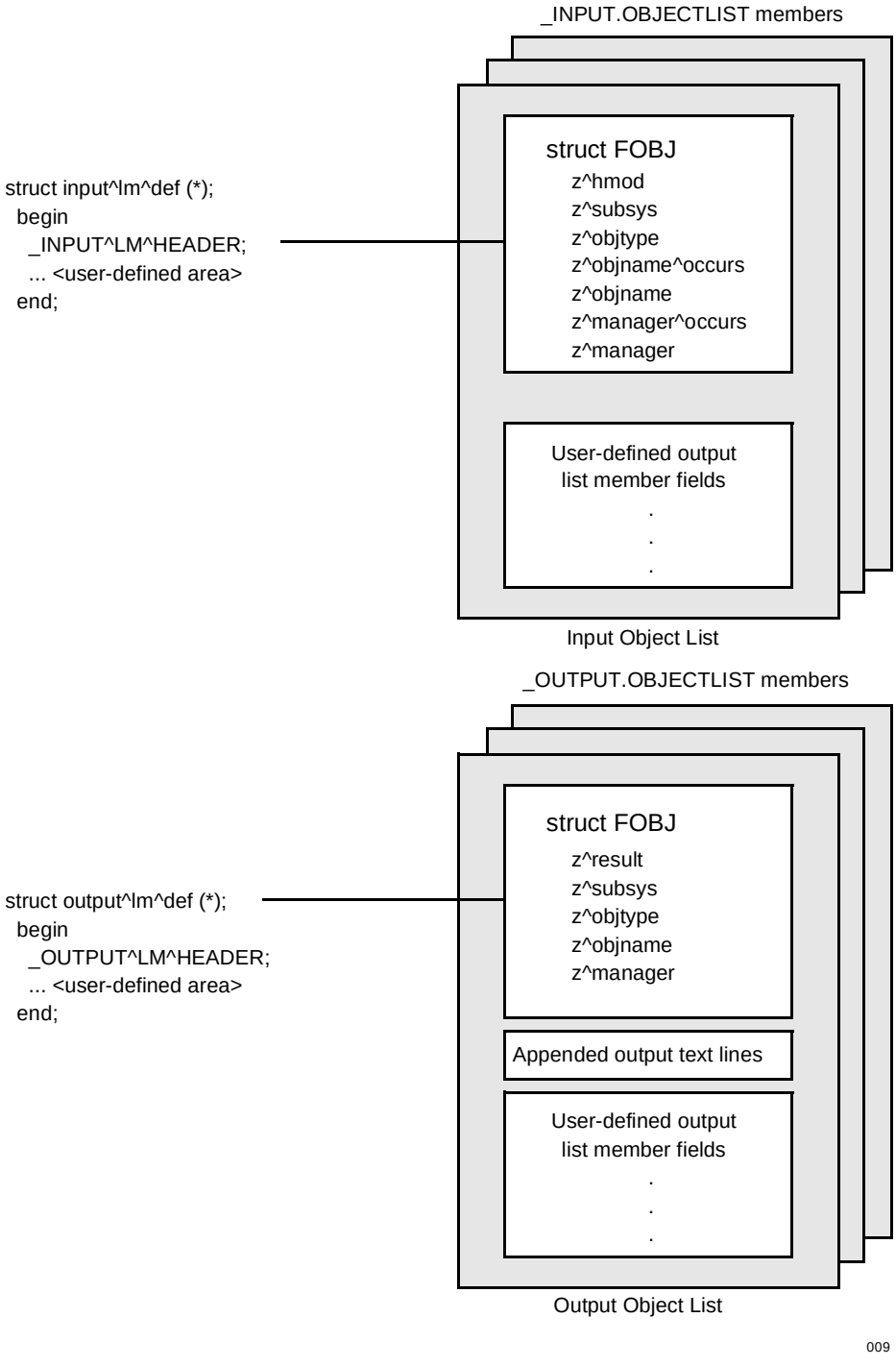
```
INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT.EXT out (_OUTPUT^DEF) := @cx._OUTPUT;
```

The Input and Output List Member Structures

In addition to defining the command context area, you must also define the input and output list member structures by specifying a structure template for each, as illustrated in Figure 3-5. The first part of each structure is reserved for use by the frame; the thread defines the rest of the structure.

Figure 3-5 shows the fields of interest to the command thread that are generated as part of the input list formatted object structure (identified as FOBJ). The figure also shows the fields that the command thread must fill in the output list member structures.

Figure 3-5. Object List Member Definitions



Defining the Input List Member Structure: _INPUT^LM^HEADER

_INPUT^LM^HEADER describes the first part of an input list member. It generates an input list formatted object structure (FOBJ). It is required as part of the input list member definition. Your _STARTUP procedure provides the frame with the length of the input list member structure

The following is an example of an input list member structure declaration:

```
STRUCT input^list^member^def (*);
  BEGIN
    _INPUT^LM^HEADER;
    < user-definitions >
   ...
  END;
```

For each member in the input object list, the following FOBJ fields are available to the thread:

Z^HMOD	Contains a hierarchy modifier for the object. If present, it overrides the hierarchy modifier (Z-HMOD) value in the _INPUT.MOD.Z^HMOD field for this object only. Values are: ZDSN^HMOD^ALL ZDSN^HMOD^ONLY ZDSN^HMOD^SUBONLY
Z^SUBSYS	Is the subsystem to which the object belongs.
Z^OBJTYPE	Is the subsystem object type of the object.
Z^OBJNAME^OCCURS	Is the length of the object name.
Z^OBJNAME	Is the object name.
Z^MANAGER^OCCURS	Is the length of the manager name, if any.
Z^MANAGER	Is the name of the manager, if any.

In addition, internal information in each input list object structure is carried forward to output list object structures when they are initialized with _FOBJECT^INIT (see "Initializing Object List Members: _FOBJECT^INIT," later in this section).

These fields are described in more detail in Section 4, "DSNM Command Requirements."

Defining the Output List Member Structure: _OUTPUT^LM^HEADER

_OUTPUT^LM^HEADER describes the first part of the user-defined output list member structure (reserved for use by the frame) and generates a DSNM formatted object structure identified as FOBJ. It is required as part of the output list member definition.

The following is an example of an output list member structure definition:

```
STRUCT output^list^member^def (*);
  BEGIN
    _OUTPUT^LM^HEADER;
   ...
  END;
```

The following FOBJ fields must be filled in by the command thread for each object in the output object list according to the specifications for the individual commands (described in Section 4, “DSNM Command Requirements”):

Z^RESULT	Contains the result code for the object in the response buffer. It may be a ZDSN^ERR value (see Appendix B, “DSNM Error Codes”), a ZDSN^STATE value, or null (zero).
Z^SUBSYS	Is the subsystem to which the object belongs.
Z^OBJTYPE	Is the subsystem object type of the object.
Z^OBJNAME	Is the object name, blank-filled.
Z^MANAGER	Is the name of the manager, if any, blank-filled.

Note. Z^OBJNAME^OCCURS and Z^MANAGER^OCCURS are present in the output object structure, but they need not be filled in.

Output objects may have lines of text associated with them as well (see “Adding Text Items to an Output Object: _APPEND^OUTPUT,” later in this section).

These fields are described in more detail in Section 4, “DSNM Command Requirements.”

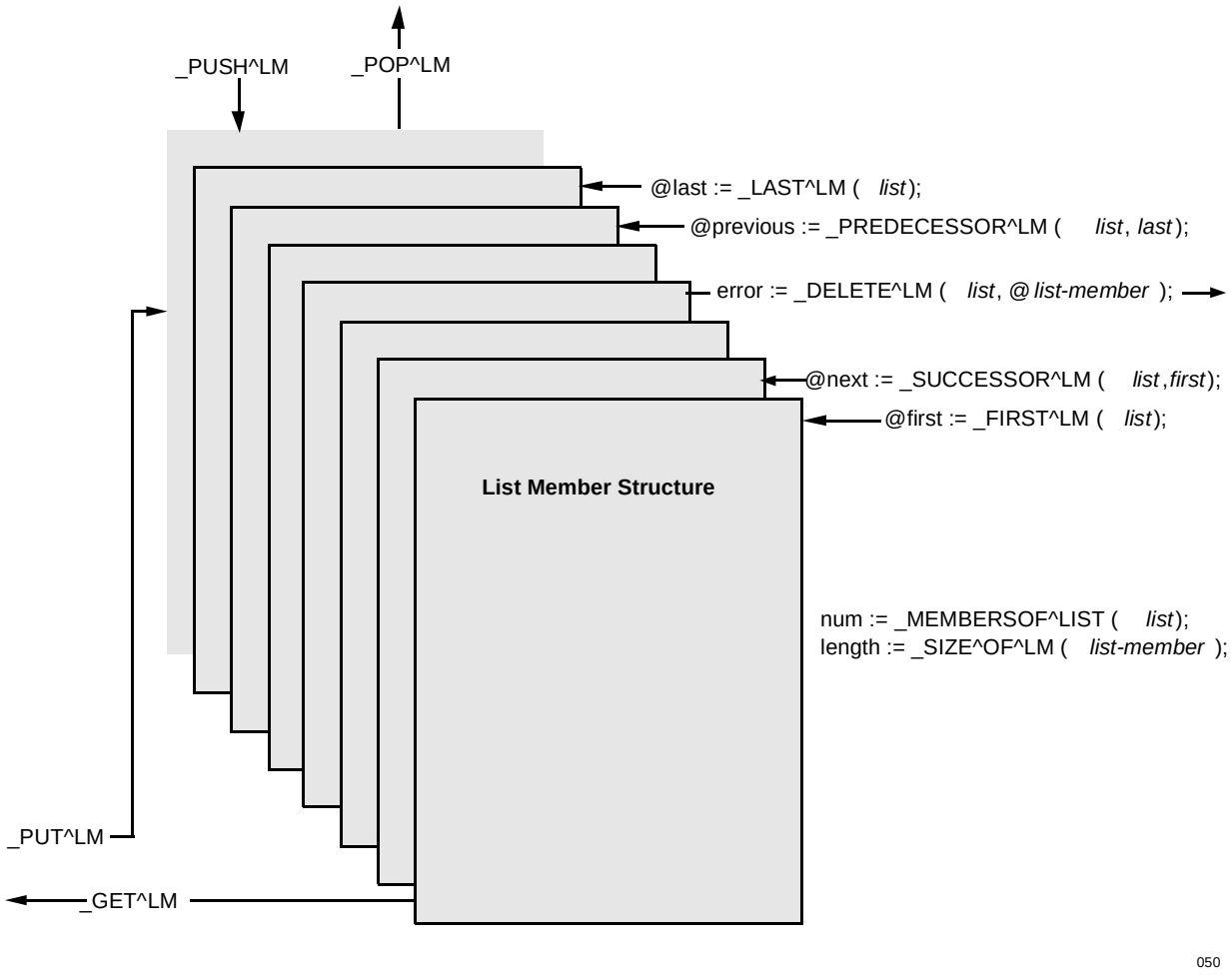
Working With Lists

The thread's overall task is to take a command and the input list of objects, and transform them into an output list of objects with associated states and/or text. The two predefined list structures available in the command context area are:

- The input object list, extracted by the frame from the command buffer.
- The output object list, filled in by the thread as a result of processing the command; it is initially empty.

As shown in Figure 3-6, list members are logically ordered. The first member is the earliest item placed on the list; the last member is the latest. Each member has a successor and a predecessor, the predecessor of the first and the successor of the last being _NULL.

Figure 3-6. Logical View of a List



Declaring a List: _LIST

Use _LIST to declare a list structure.

```
_LIST ( list );
```

Initializing a List Structure: _INITIALIZE^LIST

Use _INITIALIZE^LIST to set a list structure to nulls.

```
CALL _INITIALIZE^LIST ( list );
```

Accessing the First Member of a List: _FIRST^LM

Use _FIRST^LM to retrieve the address of the first member of a list.

```
@first-list-member := _FIRST^LM ( list );
```

Accessing the Last Member of a List: _LAST^LM

Use _LAST^LM to retrieve the address of the last (most recent) member of a list.

```
@last-list-member := _LAST^LM ( list );
```

Accessing the Next List Member: _SUCCESSOR^LM

Use _SUCCESSOR^LM to retrieve the address of the next list member in first-to-last (earliest to most-recent) order.

```
@next-list-member := _SUCCESSOR^LM ( list
                                   ,list-member );
```

Accessing the Previous List Member: _PREDECESSOR^LM

Use _PREDECESSOR^LM to retrieve the address of the previous list member in first-to-last (earliest to most-recent) order.

```
@prev-list-member := _PREDECESSOR^LM ( list
                                     ,list-member );
```

Declaring a Pointer to a List: _LISTPOINTER

Use _LISTPOINTER to declare an extended pointer to a _LIST-generated list structure.

```
_LISTPOINTER ( list );
```

Once a list pointer has been initialized with a list address, it may be used anywhere a _LIST may be used. For example:

```
INT .EXT cx(command^context^def) = _THREAD^CONTEXT^ADDRESS;
_LISTPOINTER (outlist) := @cx._OUTPUT.OBJECTLIST;
INT .EXT out^lm (output^lm^def);
   .
   .
   .
IF _ISNULL (@out^lm := _PUT^LM (outlist,,$LEN (out^lm)))
  THEN ... <out of memory> ;
   .
   .
```

Scanning a List

The following examples illustrate methods of scanning lists:

- This example scans a list forward:

```
@lm := _NULL;
WHILE _NOTNULL (@lm := _SUCCESSOR^LM (list,lm) DO
  BEGIN
  ...
  END;
```

- In the next example, the user waits for a new last member to be added to the end of a list by keeping a previous member pointer. After finding a _NULL, @lm is returned to its previous setting. @lm can be used later in _SUCCESSOR^LM to retrieve a new later member, if one has been added, or another _NULL, if one has not been added.

```
@lm := @nextlm := _NULL;
WHILE _NOTNULL (@lm := _SUCCESSOR^LM (list,lm) DO
  BEGIN
    @nextlm := @lm;
    ....
  END;
@lm := @nextlm;
```

Processing a List

Normally, you process a list either by _PUT^LM plus _GET^LM or by _PUSH^LM plus _POP^LM, but not both. _PUT^LM is identical to _PUS^†HLM, providing different sets of primitives for first-in, first-out (FIFO) and last-in, first-out (LIFO) processing, respectively. Adding a list member (_PUT^LM or _PUSH^LM) allocates new memory for the member.

Removing a member (_GET^LM or _POP^LM) does not deallocate memory immediately: the member's memory remains allocated and its contents usable until the next successive member is removed from the same end of the list, or a new member is added to the same end of the list. The removed member does not participate in list scans with _SUCCESSOR^LM or _PREDECESSOR^LM.

_UNPOP^LM and _UNGET^LM replace the last list member removed from a list with _POP^LM or _GET^LM, respectively.

First-In First-Out Processing: _PUT^LM / _GET^LM

Use _PUT^LM to allocate memory for a new last member of a list. Use _GET^LM to remove the current first member from a list (the earliest member put on the list).

```
@list-member := _PUT^LM ( list
                        ,[ length ]
                        ,initlength
                        ,[ initdata ] );

@list-member := _GET^LM ( list
                        ,[ length ] );
```

Last-In First-Out Processing: _PUSH^LM / _POP^LM

Use _PUSH^LM to allocate memory for a new last member of a list. Use _POP^LM to remove the current last member from a list (the most recent member put on the list).

```
@list-member := _PUSH^LM ( list
                         ,[ length ]
                         ,initlength
                         ,[ initdata ] );

@list-member := _POP^LM ( list
                        ,[ length ] );
```

_PUSH^LM deallocates and reuses the memory assigned to the last element removed by _POP^LM.

Maintaining a List

Use the following library services to delete list members or to join lists.

Deleting a List Member: _DELETE^LM

Use _DELETE^LM to delete a member of a list. Deleting a member removes it from the list and deallocates its memory immediately; *list-member* is set to null.

```
error := _DELETE^LM ( list
                    ,@list-member );
```

Deleting All Members of a List: _DEALLOCATE^LIST

Use _DEALLOCATE^LIST to delete all members of a list. Memory for the list members is deallocated immediately.

```
CALL _DEALLOCATE^LIST ( list );
```

Joining Two Lists: _JOIN^LIST

Use _JOIN^LIST to append all members of a source list to a destination list. When two lists are joined, data is not moved in memory. The source list is empty afterwards.

```
error := _JOIN^LIST ( dest-list
                     ,source-list );
```

Requesting Status About a List

Use the following library services to get information about lists.

Determining if a List is Empty: _EMPTY^LIST

_EMPTY^LIST is a Boolean value that is TRUE if *list* has no members.

```
_EMPTY^LIST ( list )
```

Determining the Number of List Members: _MEMBERSOF^LIST

_MEMBERSOF^LIST is the type INT(32) number of members currently in a list.

```
_MEMBERSOF^LIST ( list )
```

Initializing Object List Members: _FOBJECT^INIT

Every subsystem object processed by DSNM is defined by the contents of a ZDSN^DDL^FOBJECT^DEF structure, known as a “formatted object” or “FOBJECT structure.” The FOBJECT structure contains fields used directly by the command thread (and internal fields used by the I process frame and libraries).

It is important that every object processed by the command thread be represented in a properly initialized FOBJECT structure. Objects on the input list sent to the command thread by the frame are correctly initialized at the time the command thread is first dispatched.

Each object on an intermediate list or on the output list must also be represented in an FOBJECT structure that has been correctly initialized from a previously initialized source FOBJECT.

The source FOBJECT structure may define the same object that the new FOBJECT structure defines, or a parent object from which a new object has been derived. “Parent” here means the parent of the new object in a name hierarchy, which includes the subsystem hierarchy and an asterisk (*) object name, if supported by your I process.

You can produce new objects from objects on the input list in two ways:

1. The input object is a subsystem object, and new object names are subordinate objects produced as a result of processing a hierarchy modifier (HMOD).
2. The input object is an wild card (*), and new object names are produced as a result of expanding the wild card.

In either case, the input object is the parent of the new object in the name hierarchy (which includes the subsystem hierarchy).

Note. Outside the I process, there are higher levels possible in the name hierarchy made up of (possibly nested) DNS groups and composites.

_FOBJECT^INIT initializes a new FOBJECT structure and determines required fields from its source FOBJECT structure.

```
error := _FOBJECT^INIT ( new-fobject
                        ,[ same-fobject ]
                        ,[ parent-fobject ]);
```

One of *same-fobject* or *parent-fobject* must be supplied in the call, but not both:

- Use the *same-fobject* argument if the new FOBJECT structure is to define the same object as an existing FOBJECT structure. The new object is the same if it has the same subsystem, object type, name, and manager. Use the following syntax to initialize the new FOBJECT structure:

  ```
  error := _FOBJECT^INIT (new-fobject, same-fobject);
  ```

  The following fields from the source FOBJECT structure are copied to *new-fobject* when the *same-fobject* argument is supplied:

  Z^SUBSYS
  Z^OBJTYPE
  Z^OBJNAME^OCCURS
  Z^OBJNAME
  Z^MANAGER^OCCURS
  Z^MANAGER

- Use the *parent-fobject* argument if the new FOBJECT structure is to define a different object from any previously initialized FOBJECT structure. Specify the new object's parent in the name hierarchy; as described earlier, as *parent-fobject*. The new object is different if it differs in either object, type, or name from its "name parent" (the name from which the new object was derived by expanding a wild card or through the subsystem hierarchy). Use the following syntax to initialize the new FOBJECT structure:

  ```
  error := _FOBJECT^INIT (new-fobject,,parent-fobject);
  ```

The following fields from the source FOBJECT structure are copied to *new-fobject* when the *parent-fobject* argument is supplied:

Z^SUBSYS
Z^MANAGER^OCCURS
Z^MANAGER

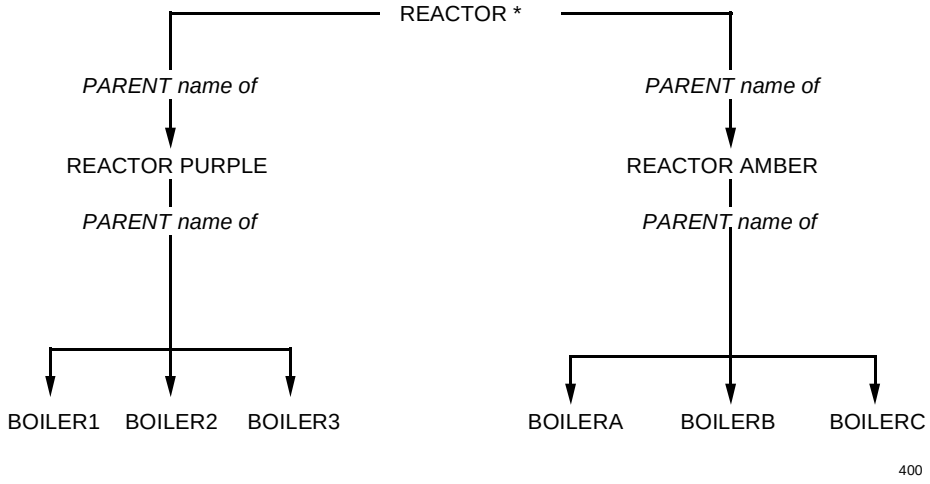
Z^OBJTYPE, Z^OBJNAME, and Z^OBJNAME^OCCURS are set to null values (zero or blanks, as appropriate). It is your responsibility to supply values for the Z^OBJTYPE and Z^OBJNAME fields. It is not necessary to fill in Z^OBJNAME^OCCURS, except for your own use.

In both cases, all required internal information is entered into the *new-fobject* structure.

Note. _FOBJECT^INIT does not allocate memory; memory for the new formatted object must be previously allocated.

The name hierarchy may extend to multiple levels. A new object may be the subordinate of an object that was in turn derived from processing a wild card object name from the original input list.

Originally, only the input list contains initialized FOBJECT structures. Every FOBJECT structure initialized with _FOBJECT^INIT must be able to be traced back to an FOBJECT structure on the original input object list, as in the following illustration (taken from the sample subsystem in Appendix D, “Sample I Process Program Code”):



Example

In the following example, an output object is initialized. The output object is derived by processing its source input object; its status, object type, and object name are filled in:

```
STRUCT input^lm^def (*);
  BEGIN
    _INPUT^LM^HEADER;
   ...
  END;

STRUCT output^lm^def (*);
  BEGIN
    _OUTPUT^LM^HEADER;
   ...
  END;

STRUCT command^context^def (*);
  BEGIN
    _COMMAND^CONTEXT^HEADER;
    INT .EXT inobj (input^lm^def);
    INT .EXT outobj (output^lm^def);
  END;

!Thread proc locals!

INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT in (input^lm^def) := @cx._INPUT;
INT .EXT out (output^lm^def) := @cx._OUTPUT;
       .
       .
       .
     ! Create output list member

IF _ISNULL (@cx.outobj := _PUT^LM (out.OBJECTLIST,,
                                   $LEN (cx.outobj)))
  THEN ... <out of available memory> ;
       .
       .
       .
   IF (error := _FOBJECT^INIT (cx.outobj.FOBJ,,
                               cx.inobj.FOBJ))
    THEN ... <error exit> ;
   cx.outobj.FOBJ.Z^RESULT := <status of subordinate>;
   cx.outobj.FOBJ.Z^OBJTYPE ':=' <type of subordinate>;
   cx.outobj.FOBJ.Z^OBJNAME ':=' <name of subordinate>;
       .
       .
       .
```

Adding Text Items to an Output Object: _APPEND^OUTPUT

For some commands, text and other variable-length items must be appended to the output object with _APPEND^OUTPUT:

```
error := _APPEND^OUTPUT ( output-list-member
                         ,type
                         ,[ header ]
                         ,[ header-len ]
                         ,[ body ]
                         ,[ body-len ] );
```

Text items are described fully under the individual command descriptions in Section 4, “DSNM Command Requirements.”

Releasing Output List Members to the Frame: _RELEASE^OUTPUT

_RELEASE^OUTPUT releases a member of the output list to the frame. Once released, the output list member can be removed by the frame at the next frame return. Each output list member should be released as soon as it has been filled in completely.

```
_RELEASE^OUTPUT ( output-list-member );
```

The frame cannot remove an output list member that has an unreleased predecessor.

Thread termination releases all output list members.

Example: List Processing Library Services

The following sample code illustrates _FOBJECT^INIT and some of the list processing library services described above. In this example, each input object and its hierarchical subordinates are to appear in the output for a STATUS command:

```
STRUCT input^lm^def (*);
  BEGIN
    _INPUT^LM^HEADER;
   ...
  END;

STRUCT output^lm^def (*);
  BEGIN
    _OUTPUT^LM^HEADER;
   ...
  END;
```

```
STRUCT command^context^def (*);
  BEGIN
    _COMMAND^CONTEXT^HEADER;
    INT .EXT inobj (input^lm^def);
    INT .EXT outobj (output^lm^def);
  END;

!Thread proc locals!

INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT in (input^lm^def) := @cx._INPUT;
INT .EXT out (output^lm^def) := @cx._OUTPUT;
    ...

! Get the next input object

IF _ISNULL (@cx.inobj := _GET^LM (in.OBJECTLIST))
  THEN ... <No more input> ;

!Create output list member

IF _ISNULL (@cx.outobj  := _PUT^LM (out.OBJECTLIST,,
                                    $LEN (cx.outobj)))
  THEN ... <out of available memory> ;

! Now cx.inobj.fobj and cx.outobj.fobj are the current
! input and output objects.  Since the output object is the
! same as an input object, use the same-fobject parameter:

IF (error := _FOBJECT^INIT (cx.outobj.FOBJ,cx.inobj.FOBJ))
  THEN ... <error exit> ;

! Send to CI, determine status of input object and its
! subordinates.
! Use state variables to return to this point after the
! _EV^IODONE event occurs.

cx.outobj.FOBJ.Z^RESULT := <status of input object>;

! Since this completes the current output object, release it

_RELEASE^OUTPUT (cx.outobj);

! Enter subordinates and their status into output list
! (Assuming one CI communication returns all subordinates)

WHILE <more subordinate objects>
 DO
  BEGIN
   IF _ISNULL (@cx.outobj :=  _PUT^LM (out.OBJECTLIST,,
                                       $LEN (cx.outobj)))
      THEN ... <out of available memory> ;
  ! Next output object

  ! Since the output object is not the same as the input
  ! object, use the parent-fobject parameter:
```

```
 IF (error :=  _FOBJECT^INIT (cx.outobj.FOBJ,,
                              cx.inobj.FOBJ))
  THEN ... <error exit> ;
 cx.outobj.FOBJ.Z^RESULT := <status of subordinate>;
 cx.outobj.FOBJ.Z^OBJTYPE ':=' <type of subordinate>;
 cx.outobj.FOBJ.Z^OBJNAME ':=' <name of subordinate>;
 _RELEASE^OUTPUT (cx.outobj);
 ...
END;
```

Suspending and Dispatching Thread Procedures

The command thread must periodically suspend execution until something occurs; then it continues at the point it left off. To temporarily suspend execution, such as for CI I/O, the thread returns an _RC^WAIT return code to the frame.

The driving mechanism for dispatching a thread is the occurrence of an event, at which point the frame calls the current thread procedure, which is entered at the top. (Any procedure that is a candidate to be dispatched as part of a thread must be declared with _THREAD^PROC and _END^THREAD^PROC.)

The command thread may return to the frame for the express purpose of having a new thread procedure dispatched (see “State Management,” later in this section).

Thread procedures may also call utility procedures, which are not thread procedures.

Suspending Thread Procedures: Return Codes

When a thread procedure cannot or should not proceed, it returns one of the following return codes to the frame:

_RC^WAIT	Redispatch the current thread procedure on the next event.
_RC^STOP	The command completed normally.
_RC^ABORT (*error*)	The command terminated abnormally. *error* is a ZDSN^ERR value indicating the reason for the abnormal command termination. See Appendix B, “DSNM Error Codes.”

Note. The library functions _DISPATCH^THREAD, _SAVE^THREAD^AND^DISPATCH, and _RESTORE^THREAD^AND^DISPATCH also result in a return to the frame with an _RC^WAIT return code. See “State Management” for more information on library functions.

Dispatching Thread Procedures: Events

A thread dispatch is initiated by an event, such as the completion of a CI communication. Events can be generated by the frame or by the thread itself:

- Events generated by the frame occur singly, with one dispatch per event.
- Events generated by the thread occur together, immediately after the next return to the frame and before any frame-generated events.

Frame-Generated Service Completion Events

The frame generates one of the following service completion events when it completes a request from the thread:

_EV^IODONE	Generated when I/O initiated by a _SEND^CI request completes
_EV^TIMEOUT	Generated when a timeout interval set by a call to _SET^TIMEOUT elapses

Frame-Generated Internal Events

The frame can generate internal events, not due directly to a request for service:

_EV^STARTUP	Generated on the frame's initial dispatch of the thread
_EV^CONTINUE	Generated when the thread returns with an _RC^WAIT and no outstanding requests are needed
_EV^CANCEL	Generated when the frame receives a command cancellation request

Examples of internal events are an event requesting cancellation of the command in progress, or an event causing the thread to be redispatched (if it returns to the frame without a pending outstanding event).

Thread-Generated Events: _SIGNAL^EVENT

The library procedure _SIGNAL^EVENT allows the thread to generate its own events. The thread can generate private events or events simulating any frame event. When the thread generates its own event(s), it is redispatched immediately when it returns _RC^WAIT to the frame.

```
CALL _SIGNAL^EVENT ( event(s) );
```

The thread may generate multiple simultaneous events with _SIGNAL^EVENT. All events signaled by the thread before RETURN _RC^WAIT appear in _LAST^EVENTS and _REAL^LAST^EVENTS together at the next thread dispatch. No frame events can appear in this case.

_LAST^EVENTS and _REAL^LAST^EVENTS are defined in "State Management."

Declaring Private Thread Events: _PRIVATE^THREAD^EVENT

_PRIVATE^THREAD^EVENT declares events, the values of which are different from any frame-generated event values, with meanings private to the thread.

```
_PRIVATE^THREAD^EVENT ( num );
```

num is a number in the range 0 through 7.

For example:

```
LITERAL next^object = _PRIVATE^THREAD^EVENT (0);
LITERAL sub^object = _PRIVATE^THREAD^EVENT (1);

CALL _SIGNAL^EVENT (sub^object + next^object);
RETURN _RC^WAIT;
   .
! After the next dispatch ...
   .
IF _ALLON (_LAST^EVENTS, sub^object + next^object)
   THEN ...;
```

Simulating Frame-Generated Events

You may simulate any frame event by signaling it with _SIGNAL^EVENT. For example:

```
CALL _SIGNAL^EVENT (_EV^IODONE);
```

Note. When you simulate a frame event, be careful not to use control variables set by frame-generated events (such as _LAST^CI^ID or _LAST^TIMEOUT^TAG), unless they are set to match the event simulated.

Declaring Utility Procedures: _RC^TYPE

Thread procedures may call utility procedures, which are not themselves thread procedures. It may be useful for such a procedure to return a valid frame return code as a function value. Use _RC^TYPE to declare:

- Function procedures that can be called by a thread procedure (but which are not themselves thread procedures) and that return a frame return code value.
- Variables to hold the frame return code (_RC^) values returned by _RC^TYPE function procedures.

```
_RC^TYPE PROC procname ;   |

_RC^TYPE var1 ,[ var2 [,...]];
```

A special return code, _RC^NULL, may be returned by an _RC^TYPE procedure to indicate that it has not returned any valid frame return code. _RC^NULL must not be returned to the frame.

In the following example, a thread procedure calls an _RC^TYPE procedure. The called procedure returns a frame return code, which is interpreted by the calling procedure.

```
_RC^TYPE PROC process^object ( ... );
  BEGIN
    .
    .
  END;

_THREAD^PROC (_COMMAND^PROC);
  BEGIN
    _RC^TYPE obj^rc;
      .
      .
      obj^rc := process^object ( ... );

      IF obj^rc <> _RC^NULL
        THEN
          RETURN obj^rc;
      .
      .
  _END^THREAD^PROC;
```

State Management

As described earlier, each thread is allocated a context space when created. The context space exists until the thread terminates. The command context space and all dynamically allocated memory areas are preserved between dispatches of the thread.

Local variables are not preserved between dispatches and must be reinitialized after any dispatch before they are used. Global variables are shared among all concurrently executing threads. There is no way for the user to order dispatching among active concurrent threads; therefore, only read-only globals are practical as a general rule.

When the frame dispatches a thread, the current thread procedure is always entered from the top. It is up to the thread procedure to determine its current state from the event that occurred and from information it has kept in its command context area.

The frame maintains the following state variables:

- Event(s) that caused the current dispatch
- Current thread state
- Current thread procedure

Determining Which Event(s) Caused the Current Dispatch

_REAL^LAST^EVENTS and _LAST^EVENTS allow the thread to determine which event(s) caused the current dispatch.

_REAL^LAST^EVENTS

Each time the command is dispatched, _REAL^LAST^EVENTS is set to contain the event(s) that caused the current dispatch. Each bit represents a different event. _REAL^LAST^EVENTS is a define that returns a value; therefore it can only be tested, not altered.

```
_REAL^LAST^EVENTS
```

For example, to see if the thread is dispatched by a request to cancel the command:

```
IF _ON (_REAL^LAST^EVENTS, _EV^CANCEL)
  THEN ...;
```

Only one frame event occurs with one dispatch per event, so only one bit of _REAL^LAST^EVENTS is ever on for a frame event.

The thread may generate multiple simultaneous events with _SIGNAL^EVENT. All events signaled by the thread before _RC^WAIT appear in _LAST^EVENTS at the next thread dispatch. No frame-generated events can appear in this case.

_LAST^EVENTS

Each time the command is dispatched, _LAST^EVENTS is set to contain the event(s) that caused the current dispatch. _LAST^EVENTS is a global variable that can be altered as well as tested.

```
_LAST^EVENTS
```

For example, since a command that terminates early due to a cancel event from the frame is considered to have terminated normally, you might want to treat _EV^CANCEL as _EV^IODONE by altering the contents of _LAST^EVENTS:

```
_TURNOFF (_LAST^EVENTS, _EV^CANCEL);
_TURNON (_LAST^EVENTS, _EV^IODONE);
```

Note. When a thread is invoked for the first time, _LAST^EVENTS and _REAL^LAST^EVENTS are set to _EV^STARTUP.

Altering the Current Thread Procedure and Thread State

When the frame dispatches the thread, it calls the current thread procedure. You can alter the current thread procedure (and, in some cases, the thread state also) to be called by the frame at the next thread dispatch by using any one of the following procedures:

_SET^THREAD^PROC
_THREAD^STATE
_PUSH^THREAD^PROCSTATE
_POP^THREAD^PROCSTATE
_DISPATCH^THREAD
_SAVE^THREAD^AND^DISPATCH
_RESTORE^THREAD^AND^DISPATCH

Altering the current thread procedure is a high-level state change. As shown in the following example, the initial thread procedure might examine a command passed to it by the frame when it is first dispatched. The thread procedure determines if the command is an informational or state-change command. Since these two types of commands have considerably different output requirements, it may be convenient to have different procedures perform their processing.

```
_THREAD^PROC (info^thread^proc);
  BEGIN
   < procedure body >
  _END^THREAD^PROC;

_THREAD^PROC (state^change^thread^proc);
  BEGIN
   < procedure body >
  _END^THREAD^PROC;

_THREAD^PROC (_COMMAND^PROC);
  BEGIN
    .
    .
    IF info-type-command
      THEN _SET^THREAD^PROC (@info^thread^proc)
      ELSE _SET^THREAD^PROC (@state^change^thread^proc);
    CALL _SIGNAL^EVENT (_EV^STARTUP);
    RETURN _RC^WAIT;
    .
    .
  _END^THREAD^PROC;
```

Setting the Current Thread Procedure: _SET^THREAD^PROC

_SET^THREAD^PROC allows you to set the thread procedure to be called by the frame at the next thread dispatch.

```
_SET^THREAD^PROC ( @procname );
```

Determining and Setting the Current Thread State: _THREAD^STATE

The frame sets the thread state to _ST^INITIAL when it creates a thread. Subsequently, you may set the thread state as desired; the frame never uses it again. The current thread state can be tested or set with _THREAD^STATE.

The following example tests the current state of the thread:

```
CASE _THREAD^STATE OF
  BEGIN
    _ST^INITIAL  ->
      ...
    OTHERWISE ->
      ...
  END;
```

Currently, _ST^INITIAL is the only reserved thread state value.

Defining Thread States: _ST^MIN^THREAD^STATE

Thread state values are always nonnegative. The literal _ST^MIN^THREAD^STATE is the minimum value to which a user-defined thread state can be set. Use this literal to define thread states.

The following example declares several thread states and then sets the current thread state:

```
LITERAL thr^state1 = _ST^MIN^THREAD^STATE, thr^state2,
        thr^state3;
  .
  .
  .
_THREAD^STATE := thr^state2;
```

Saving and Restoring Current Thread Procedure and State Values: _PUSH^THREAD^PROCSTATE and _POP^THREAD^PROCSTATE

_PUSH^THREAD^PROCSTATE and _POP^THREAD^PROCSTATE allow you to save and restore the current thread procedure and thread state.

```
error := _PUSH^THREAD^PROCSTATE ( [ @procname ] ,[ state ] );
error := _POP^THREAD^PROCSTATE ;
```

_PUSH^THREAD^PROCSTATE saves the current thread procedure and thread state. It optionally sets new values for the current thread procedure and thread state.

_POP^THREAD^PROCSTATE restores the saved values.

In the following example, the frame dispatches PROC^X of the command thread in _ST^INITIAL.

PROC^X calls PROC^Y in STATE^B by:

- Setting its return state to STATE^A.
- Saving the old current thread procedure and state values, and setting new current thread procedure and thread state values.
- Signaling an event and returning to the frame to dispatch the new thread procedure PROC^Y in the new state STATE^B.

PROC^Y checks for event _EV^STARTUP; resets the current thread procedure and thread state to the previously saved values of PROC^X and STATE^A; and returns to the frame to dispatch PROC^X in STATE^A.

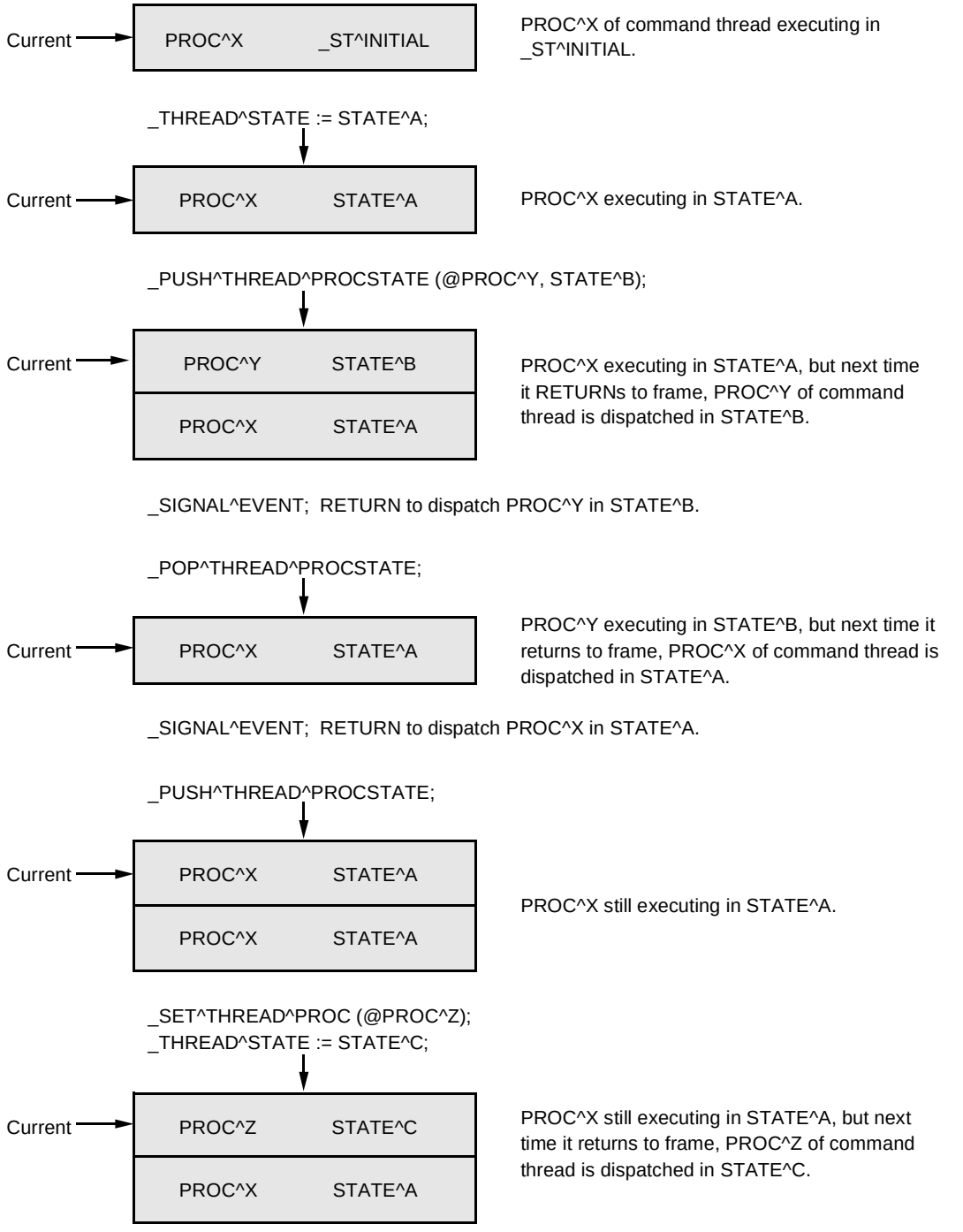
```
_THREAD^PROC (PROC^X);
  BEGIN
  .
  CASE _THREAD^STATE OF
    BEGIN
      _ST^INITIAL -->
        _THREAD^STATE := STATE^A;
        IF (error :=_PUSH^THREAD^PROCSTATE (@PROC^Y,STATE^B))
          THEN ... <error> ;
        CALL _SIGNAL^EVENT (_EV^STARTUP);
        RETURN _RC^WAIT;

      STATE^A -->
          .
          .
        RETURN _RC^STOP;
    END;
  _END^THREAD^PROC;‘

_THREAD^PROC (PROC^Y);
  BEGIN
  .
  CASE _THREAD^STATE OF
    BEGIN
      STATE^B -->
        IF _ON (_LAST^EVENTS,_EV^STARTUP)
          THEN
            BEGIN
               .
               .
              IF (error := _POP^THREAD^PROCSTATE)
                THEN ... <error> ;
              CALL _SIGNAL^EVENT (_EV^CONTINUE);
              RETURN _RC^WAIT;
            END;
    END;
  _END^THREAD^PROC;
```

Figure 3-7 illustrates using _THREAD^STATE, _SET^THREAD^PROC, _PUSH^THREAD^PROCSTATE, and _POP^THREAD^PROCSTATE.

Figure 3-7. Altering Current Thread Procedure and Thread State Values



Dispatching a New Thread Procedure: _DISPATCH^THREAD

_DISPATCH^THREAD returns to the frame and causes a new dispatch. _DISPATCH^THREAD does not save any information about the procedure from which it was invoked.

```
_DISPATCH^THREAD ([ @procname ]
                 ,[ state ]
                 ,[ event ]);
```

Saving Context and Dispatching a New Thread Procedure: _SAVE^THREAD^AND^DISPATCH

_SAVE^THREAD^AND^DISPATCH saves the current thread procedure and state, optionally sets new thread procedure and state values, and returns to the frame for immediate dispatch with the specified event (or _EV^CONTINUE if none specified).

```
_SAVE^THREAD^AND^DISPATCH ([ @procname ]
                          ,[ state ]
                          ,[ event ]);
```

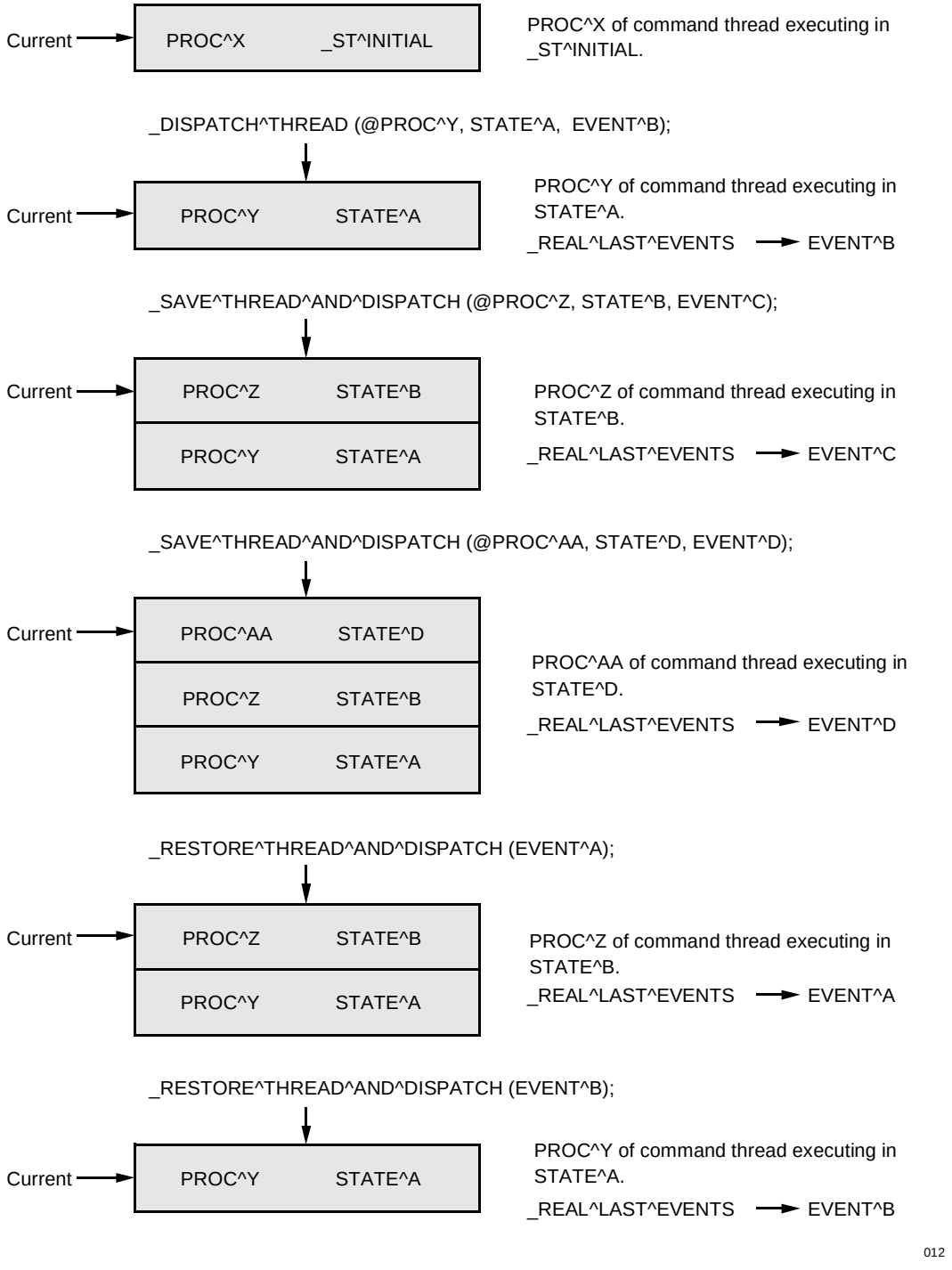
Restoring and Dispatching Previous Context: _RESTORE^THREAD^AND^DISPATCH

_RESTORE^THREAD^AND^DISPATCH restores the thread procedure and state last saved, and dispatches it with the specified event (or _EV^CONTINUE if none specified).

```
_RESTORE^THREAD^AND^DISPATCH ([ event ]);
```

Figure 3-8 illustrates using _SAVE^THREAD^AND^DISPATCH, _RESTORE^THREAD^AND^DISPATCH, and _DISPATCH^THREAD.

Figure 3-8. Dispatching New Thread Procedures



Frame Services

The frame transmits CI messages and sets timeout intervals for the command thread.

To request a frame service, such as CI communication, the command thread calls a library procedure and eventually returns to the frame to wait for the event signaling the completion of the service.

The frame generates a service completion event (_EV^IODONE or _EV^TIMEOUT) at the completion of each service and then redispatches the thread.

The thread can initiate more than one service request before returning to the frame.

CI Communications

Communicating with a CI involves the following steps:

1. Declaring a global pointer to a structure (defined by the template _CI^DEF) for each CI with which your I process communicates.
2. Retrieving CI configuration parameters from the DSNM configuration by calling _ADD^CI in your _STARTUP procedure. The frame uses this information when it opens a CI for communication. _ADD^CI allocates the memory for, fills in, and returns a pointer to the _CI^DEF-defined structure.
3. Declaring a CIID structure with _CI^ID in which information about an open CI is stored. (You may also find it convenient to declare an extended pointer to that structure with _CI^IDPOINTER.)
4. Opening the CI for communication (_OPEN^CI).
5. Sending request buffer(s) to the CI (_SEND^CI).
6. Terminating the CI communication (_CLOSE^CI).

The CI Configuration Structure: _CI^DEF

_CI^DEF is a template for a CI configuration structure that is filled in by the _ADD^CI procedure. You declare an extended pointer to the structure in globals; for example:

```
INT .EXT ci^config (_CI^DEF);
```

Note. The _CI^DEF-defined CI configuration structure plays a role in command thread CI communications analogous to file name in NonStop Kernel interprocess communications. A CI is identified by its CI configuration name.

The definition of the _CI^DEF-defined structure is:

```
DEFINITION ZDSN-DDL-PCLASS-CONFIG.
  02 Z-PCLASS               TYPE ZDSN-DDL-PCLASS.
  02 Z-PUBLIC-NAME-OCCURS   TYPE ZSPI-DDL-UINT.
  02 Z-PUBLIC-NAME          TYPE ZDSN-DDL-PARAMNAME.
  02 Z-FLAGS                TYPE ZSPI-DDL-ENUM.
  02 Z-PNAME-OCCURS         TYPE ZSPI-DDL-UINT.
  02 Z-PNAME                TYPE ZDSN-DDL-PNAME.
  02 Z-MAX-PROCESSES        TYPE ZSPI-DDL-INT.
  02 Z-OPEN-PARAMS.
    03 Z-DEFAULT-QUALIFIER TYPE ZDSN-DDL-PQUAL.
    03 Z-NOWAIT-DEPTH      TYPE ZSPI-DDL-INT.
    03 Z-OPEN-TIMEOUT      TYPE ZSPI-DDL-INT2.
  02 Z-NEWPROCESS-PARAMS.
    03 Z-OBJECT-FILE       TYPE ZDSN-DDL-OBJNAME.
    03 Z-LIBRARY-FILE      TYPE ZDSN-DDL-OBJNAME.
    03 Z-SWAPVOL           TYPE ZDSN-DDL-OBJNAME.
    03 Z-PRIORITY          TYPE ZSPI-DDL-INT.
    03 Z-DATAPAGES         TYPE ZSPI-DDL-INT.
    03 Z-NUM-CPUS          TYPE ZSPI-DDL-INT.
    03 Z-CPUS              TYPE ZSPI-DDL-INT OCCURS 16 TIMES.
    03 Z-HOMETERM          TYPE ZDSN-DDL-OBJNAME.
    03 Z-FLAGS             TYPE ZSPI-DDL-ENUM.
END
```

Retrieving CI Configuration Parameter Values: _ADD^CI

_ADD^CI allocates memory for the _CI^DEF-defined structure, fills it in with CI configuration information from the DSNM configuration, and returns the address of the filled-in structure.

```
@ci-config := _ADD^CI ( ciname
                      ,[ error ]
                      ,[ error-filename ]);
```

You must call _ADD^CI in your _STARTUP procedure for each CI class with which your I process communicates.

ciname is the process class name of the CI. The process class name is arbitrary; by custom, the object file name of the subsystem manager is the logical name of the process class: for example, PATHMON or SCP.

Note. The CI process class name is configured in the DSNMCONF file (in the COMPONENT field in the class CI-CONFIG record for this CI). See Section 6, “Configuring a New Subsystem Into DSNM,” for more information about CI configuration.

Declaring a CIID Structure: _CI^ID

To access a CI, first use _CI^ID to declare a structure, referred to as a "CIID structure." A later call to _OPEN^CI causes information about the CI to be stored in the CIID structure.

```
_CI^ID ( ciid );
```

ciid is the name (a valid TAL identifier) of the CIID structure by which an open CI can be referred.

Note. The CIID structure plays a role in command thread CI communication analogous to a file number in NonStop Kernel interprocess communications. A particular instance of an open CI is identified by its *ciid* in CI communications.

Declaring a Pointer to a CIID Structure: _CI^IDPOINTER

Use _CI^IDPOINTER to declare an extended pointer to a CIID structure.

```
_CI^IDPOINTER ( ciid );
```

Opening a CI for Communication: _OPEN^CI

_OPEN^CI opens a CI for communication.

```
error := _OPEN^CI ( ci-config
                  ,ciid
                  ,[ processname ]
                  ,[ nowait-depth ])
```

Requesting a CI Communication: _SEND^CI

The frame sends messages to the CI. Use _SEND^CI to initiate sending a message:

```
error := _SEND^CI ( ciid
                  ,buffer
                  ,write-count
                  ,reply-count
                  ,[ context-boolean ]
                  ,[ tag ]
                  ,[ timeout ] );
```

You must allocate a message buffer large enough to hold the larger of the message and its response. This buffer must be in the command context space or in an allocated list member: it cannot be in globals or locals. If more than one operation is to be outstanding (whether on the same or on separate CIs), you should also supply an INT(32) tag for the operation, usually a pointer to some identifying data.

After initiating a request for CI communication, the thread must return to the frame to wait for its completion with a RETURN _RC^WAIT.

When the communication is complete, the frame dispatches the thread with the event _EV^IODONE.

Canceling a CI Communication Request: _CANCEL^SEND^CI

Use _CANCEL^SEND^CI to cancel an outstanding CI communication request:

```
error := _CANCEL^SEND^CI ( [ tag ] );
```

The frame cancels any outstanding _SEND^CI operations when the thread terminates.

Terminating a CI Communication: _CLOSE^CI

You must close a CI before its CIID structure (*ciid*) is used in another _OPEN^CI operation. _CLOSE^CI terminates a CI communication and cancels any outstanding I/O operations.

```
error := _CLOSE^CI ( ciid );
```

Accessing Information About a CI Communication

Use the following library services to retrieve information about a CI communication:

_LAST^CI^ID	Address of CI involved in last CI communication when _EV^IODONE occurs
_CI^LASTERROR (*ciid*)	INT file system error of last operation
_CI^REPLYADDRESS (*ciid*)	INT(32) extended address of reply
_CI^REPLYLENGTH (*ciid*)	INT length of reply
_CI^REPLYTAG (*ciid*)	INT(32) tag of last operation
_CI^FILENUM (*ciid*)	INT file number of CI

The Most-Recently Completed CI Communication: _LAST^CI^ID

_LAST^CI^ID is a field within the command context space giving the CI involved in the CI communication terminated by the last _EV^IODONE event. Its type is _CI^IDPOINTER.

```
_CI^IDPOINTER (mgr);
INT .EXT cx(command^context^def) = _THREAD^CONTEXT^ADDRESS;

IF _ON (_LAST^EVENTS, _EV^IODONE)
 THEN
   BEGIN
     @mgr := cx._LAST^CI^ID;
       IF _CI^LASTERROR (mgr)  ! check for errors
         THEN ... ;
   END;
```

File System Error: _CI^LASTERROR

_CI^LASTERROR is the type INT file system error of the last CI operation (see previous example).

```
fserror := _CI^LASTERROR ( ciid )
```

Address of CI Reply Buffer: _CI^REPLYADDRESS

_CI^REPLYADDRESS is the type INT(32) extended address of the reply buffer containing information read from a CI on completion of a _SEND^CI.

```
replyaddress := _CI^REPLYADDRESS ( ciid )
```

Length of CI Reply Buffer: _CI^REPLYLENGTH

_CI^REPLYLENGTH is the type INT length of the reply buffer containing information read from a CI on completion of a _SEND^CI.

```
replylen := _CI^REPLYLENGTH ( ciid )
```

Tag of Last CI Operation: _CI^REPLYTAG

_CI^REPLYTAG is the type INT(32) tag associated with the last CI operation.

```
replytag := _CI^REPLYTAG ( ciid )
```

File Number: _CI^FILENUM

_CI^FILENUM is the type INT file number of the CI involved with the most-recently completed communication.

```
filenumber := _CI^FILENUM ( ciid )
```

Timeout Intervals

The command thread may create a pause by arranging for a future timeout event and then returning to the thread to wait for it.

Requesting a Timeout Interval: _SET^TIMEOUT

Use _SET^TIMEOUT to delay for a time interval:

```
CALL _SET^TIMEOUT ( time-interval
                  ,[ tag ] );
```

After initiating the request, the thread returns to the frame with _RC^WAIT.

When the time interval elapses, the frame dispatches the thread with the event _EV^TIMEOUT. *time-interval* establishes the time in hundredths (0.01) of a second and is a type INT(32) expression.

tag is a type INT(32) expression.

Accessing the Timeout Request Tag: _LAST^TIMEOUT^TAG

Use _LAST^TIMEOUT^TAG to access the tag associated with a timeout request:

```
_LAST^TIMEOUT^TAG
```

It is convenient to use the address of a list member as a timeout tag to hold information about the purpose of the timeout, as illustrated in the following example:

```
STRUCT time^info^def (*);
  BEGIN
    .
    .
  END;

INT .EXT time^info (time^info^def);

IF _ISNULL (@time^info := _PUT^LM (cx.worklist,,
                                   $LEN (time^info)))
  THEN ... <out of memory> ;

     < fill in time^info data >

CALL _SET^TIMEOUT (time, @time^info);
RETURN _RC^WAIT;          !Wait for _EV^TIMEOUT
```

```
    .
    .
    .
IF _ON (_LAST^EVENTS, _EV^TIMEOUT)
  THEN
    BEGIN
      @time^info := cx._LAST^TIMEOUT^TAG;
        <process time^info data >
      CALL _DELETE^LM (cx.worklist, @time^info);
    END;
```

_LAST^TIMEOUT^TAG is a type INT(32) expression.

Canceling a Timeout Request: _CANCEL^TIMEOUT

Cancel an outstanding timeout with _CANCEL^TIMEOUT:

```
error := _CANCEL^TIMEOUT ( [ tag ] );
```

tag is a type INT(32) expression.

Command Thread Termination

When the command thread terminates, the frame performs the following:

- Issues _CLOSE^CI for all open CIs.
- Issues _CANCEL^TIMEOUT for any outstanding _SET^TIMEOUT operation.
- Issues _CANCEL^SEND^CI for any outstanding _SEND^CI operation.
- Releases all output list members.
- Deallocates any remaining input object list members.

The thread must deallocate all lists containing current members other than input and output lists (_COMMAND^TERMINATION^PROC is a convenient place to do this).

Reporting Errors

When a subsystem or DSNM error occurs, the command thread should:

- Attend to outstanding CI operations. This may involve sending a sequence of commands to the CI to return it to a clean state. The frame cancels outstanding I/O operations when a thread terminates. This is sufficient for context-free server CIs.

Note. Since the error may involve running out of memory, sufficient memory should be reserved for this purpose, either in the command context space or in a working list member, before the first CI communication is initiated.

- If appropriate, generate an EMS event (see “Reporting Errors to EMS” for guidelines).
- Return to the frame with an appropriate _RC^ABORT error to describe the error.

The _EV^CANCEL event should be handled like an error except that it is a normal thread termination (return code _RC^STOP is appropriate).

Reporting Errors to the Frame

Errors that do not terminate the command must be associated with some command object (for instance, an object name unknown by the subsystem). Usually, errors associated with an object should not terminate the command, although there may be exceptions for individual subsystems.

Errors that do not terminate the command are reported in the FOBJECT result code field (Z^RESULT) of the affected output object structure. The result code must be one of the defined ZDSN^ERR token values (see Appendix B, “DSNM Error Codes”). The structure must also contain all entries appropriate for the executed command, including the fully qualified object name.

For errors generated by the subsystem, the result code should be ZDSN^ERR^SUBSYSTEM^ERR. In addition, one line of result text (ZDSN^VTY^RESULTTEXT), briefly describing the subsystem error, should be appended to the output object. The result text must not duplicate the information of the result code, but add to it. To formulate result text, assume that presentation services will substitute the text listed in Appendix B, “DSNM Error Codes,” associated with each ZDSN^ERR code in the error display.

Note. ZDSN^EMOD^SUPPRESS allows the user to suppress errors associated with the subsystem (undefined objects, unreachable managers, and so on); when ZDSN^EMOD^SUPPRESS is in effect, error objects should be omitted from the output object list.

If ZDSN^EMOD^DETAIL is in effect, and if there is more error information available from the subsystem than can reasonably be given in the one line of result text, one or more lines of error text (ZDSN^VTY^ERRTEXT) should be appended to the structure. The decision to supply error text depends upon the information available from the subsystem: it should be omitted unless there is genuine additional information to be transmitted.

Errors returned from a subsystem or arising when accessing the subsystem should be reported using the following ZDSN^ERR values:

- ZDSN^ERR^FS^ERR

  Use this value to report file system errors that occur when accessing the subsystem (for example, if the manager is not running) or that are passed through from the subsystem about a subsystem object (for example, a security violation). The file-system error number should be appended as ZDSN^VTY^RESULTTEXT.

  If the file that caused the error is not the same as the object in the object structure, the error file name should be given in the text also, as “*node*.$*filename*.”

- ZDSN^ERR^OBJ^NOT^FOUND

  Use this value to report an object unknown to the subsystem. No result text should be included.

- ZDSN^ERR^SUBSYSTEM^ERR

  Use this value to report all other subsystem errors. The error should be described as result text.

Command-Terminating Errors

If an error causes a command to terminate, the command thread must clean up its resources. Such an error should not affect commands that might be active on other threads at the time. As a general rule, an error should terminate the thread in which it occurs by returning _RC^ABORT (error) to the frame, but should not cause the I process to terminate by calling PROCESS_STOP_.

Errors that are the result of logic or data errors in the command thread, or errors in any component supplied by Tandem, should also be reported to EMS. Errors that represent normal (although infrequent) conditions, such as running out of memory, should not be reported to EMS.

Some errors, such as corrupted global data, may be so serious that there is no choice but to terminate the process. These errors should always be reported to EMS.

Reporting Errors to EMS

Use _REPORT^STARTUP^ERROR and _REPORT^INTERNAL^ERROR to log serious errors to the EMS collector process. These procedures are summarized in Table 3-1, and described in detail in Appendix A, “DSNM Library Services.”

Overview of the Library Services

Table 3-1 lists the library services that support the I process development model.

Table 3-1. Summary of I Process Development Library Services (page 1 of 6)

Function	Arguments	Description
Booleans and Bit Manipulation		
Given that: A is an INT variable and B is an INT expression; F is a bit mask INT expression specifying which bits of the other operands are affected or participate in the operation; X and Y are INT(32) expressions.		
Bit test Booleans (true/false)	_ON (B, F)	TRUE if any one-bit of F is on in B.
	_OFF (B, F)	TRUE if any one-bit of F is off in B.
	_ANYON (B, F)	TRUE if any one-bit of F is on in B.
	_ANYOFF (B, F)	TRUE if any one-bit of F is off in B.
	_ALLON (B, F)	TRUE if every one-bit of F is on in B.
	_ALLOFF (B, F)	TRUE if every one-bit of F is off in B.
Functions returning a value	_EXTRACT (B, F);	Has value of those one-bits of F that are on in B.
	_BITDEF (B ,[*max-bit*] ,[*min-bit*])	Defines a bit within a specified range.
Executable functions (no value returned)	_TURNON (A, F);	Turn on all one-bits of F in A.
	_TURNOFF (A, F);	Turn off all one-bits of F in A.
	_DEPOSIT (A, B, F);	Set bits in A equal to same bits in B as selected by one-bits in F.
Executable functions (value returned)	_ALLON^TURNOFF (A, F);	TRUE if every one-bit of F is on in A; turns off every one-bit in A that is on in F.
	_ANYON^TURNOFF (A, F);	TRUE if any one-bit of F is on in A; turns off every one-bit in A that is on in F.
Extended address Booleans	_ISNULL (X)	TRUE if X is a null extended memory pointer.
	_NOTNULL (X)	TRUE if X is a nonnull extended memory pointer.
	_XADR^EQ (X, Y)	TRUE if two valid extended addresses are equal.
	_XADR^NEQ (X, Y)	TRUE if two valid extended addresses are not equal.
Defining Objects		
Generating formatted object structure	_INPUT^LM^HEADER ;	Generates formatted object portion of an input list member structure.
	_OUTPUT^LM^HEADER ;	Generates formatted object portion of an output list member structure.
Initializing	*error* := _FOBJECT^INIT (*new-fobject* ,[*same-fobject*] ,[*parent-fobject*]);	Initializes new object.

Table 3-1. Summary of I Process Development Library Services (page 2 of 6)

Function	Arguments	Description
Appending text	*error* := _APPEND^OUTPUT (*output-list-member*, *type* ,[*header*] ,[*header-len*] ,[*body*] ,[*body-len*]);	Appends text and other variable-length items to an output object.
Releasing	_RELEASE^OUTPUT (*output-list-member*);	Releases a member of the output list to the frame.
User-Written Procedure Declarations		
Startup procedures	INT PROC _STARTUP (*context-len* , *input-lm-len*) EXTENSIBLE;	Supplies lengths of user context area and input list members, and retrieves subsystem and CI configuration parameters for frame.
	INT PROC _STARTUP^MODE (*component*, *testmode*, *accept-startup-component*, *subject*) EXTENSIBLE;	Supplies startup processing information to frame.
	_COMPILED^IN^TESTMODE	Literal set to 1 (TRUE) if source file is compiled in test mode and 0 otherwise. Used to set *testmode* parameter value in _STARTUP^MODE procedure
Thread procedures	_THREAD^PROC (*procname*); _END^THREAD^PROC;	Declares any procedure that can be dispatched by the frame.
Initial command thread procedure	_THREAD^PROC (_COMMAND^PROC); _END^THREAD^PROC;	Name of initial command thread procedure.
Thread termination procedure	_THREAD^TERMINATION^PROC (COMMAND^TERMINATION^PROC); _END^THREAD^TERMINATION^PROC;	Declares thread termination procedure.
Command thread utility procedures	_RC^TYPE *procname* ; _RC^TYPE *var1* [,*var2* , [...]] ;	Declares procedures that return a valid frame return code value but are not themselves thread procedures. May also be used to declare variables that hold frame return code values.
	_RC^NULL	Special return code that may be returned by a utility procedure that was called by an _RC^TYPE thread utility procedure, indicating that the procedure has not returned any valid frame return code.
Configuration		
Parameter retrieval structures	_CI^DEF	Defines a CI configuration structure to be filled in by _ADD^CI.
	_SUBSYS^DEF	Defines a subsystem configuration structure to be filled in by _ADD^SUBSYS.

Table 3-1. Summary of I Process Development Library Services (page 3 of 6)

Function	Arguments	Description
Parameter retrieval procedures	@*ci-config* := _ADD^CI (*ciname* ,[*error*] ,[*error-filename*]);	Fills in _CI^DEF-defined structure with CI configuration information.
	@*ss-config* := _ADD^SUBSYS (*ssname* ,[*error*] ,[*error-filename*]);	Fills in _SUBSYS^DEF-defined structure with subsystem and object type configuration information.
	error := _GET^PARAM (*scope* , *type* ,[*subsys*] ,[*class*] ,[*component*] , *paramname* , *paramvalue*:*maxlen* ,[*len*] ,[*error-filename*]);	Retrieves a DSNM configuration parameter.
	error := _GET^PROCESS^PARAM (*paramname* , *paramvalue*:*maxlen* ,[*len*]);	Retrieves a process startup parameter.
Thread Procedure Control Flow		
Return codes	RETURN _RC^WAIT;	Redispatches thread on next event.
	RETURN _RC^STOP;	Command completed normally.
	RETURN _RC^ABORT (*error*);	Command terminated abnormally.
Frame events	_EV^CANCEL	Cancel the current command.
	_EV^CONTINUE	Default event when no other event can occur.
	_EV^IODONE	_SEND^CI request completed.
	_EV^STARTUP	Initial dispatch of thread after thread is created.
	_EV^TIMEOUT	Timeout interval has elapsed.
Thread events	CALL _SIGNAL^EVENT (*event(s)*);	Lets thread generate its own event and be redispatched immediately upon return to frame.
	_PRIVATE^THREAD^EVENT (*num*);	Declares event different from any frame-generated event.
Testing and altering events	_LAST^EVENTS	Tests or alters event(s) that caused current thread dispatch.
	_REAL^LAST^EVENTS	Determines event(s) that caused current thread dispatch.
List Processing		
Declarations	_LIST (*list*);	Declares a list.
	_LISTPOINTER (*list*);	Declares extended pointer to a list.
Initializing	CALL _INITIALIZE^LIST (*list*);	Sets a list structure to nulls.
Scanning	@lm := _FIRST^LM (*list*);	Points to first member of list.
	@lm := _LAST^LM (*list*);	Points to last member of list.
	@lm := _SUCCESSOR^LM (*list*, *list-member*);	Points to next member in first-to-last order.
	@lm := _PREDECESSOR^LM (*list*, *list-member*);	Points to next member in last-to-first order.

Table 3-1. Summary of I Process Development Library Services (page 4 of 6)

Function	Arguments	Description
Processing	@lm := _PUT^LM (*list* ,[*length*] , *initlength* ,[*initdata*]); @lm := _GET^LM (*list* ,[*length*])	FIFO processing (first in, first out): adds new last member; removes current first member.
	@lm := _PUSH^LM (*list* ,[*length*] , *initlength* ,[*initdata*]); @lm := _POP^LM (*list* ,[*length*]);	LIFO processing (last in, first out): adds new last member; removes current last member.
	error := _UNGET (*list* , *list-member*); *error* := _UNPOP (*list* , *list-member*);	Replaces last list member removed from a list.
Maintenance	*error* := _DELETE^LM (*list*, @*list-member*);	Deletes any member of list.
	CALL _DEALLOCATE^LIST (*list*);	Deallocates all members of list.
	error := _JOIN^LIST (*list1*, *list2*);	Concatenates lists.
Returns information about a list	IF _EMPTY^LIST (*list*) THEN ...	TRUE if list has no members.
	num := _MEMBERSOF^LIST (*list*);	Number of current list members.
State Management		
Altering current thread procedure/ thread state	_SET^THREADPROC (@*procname*);	Sets current thread procedure to be called at next thread dispatch.
	_THREAD^STATE	Current thread state; may be tested or altered.
	_DISPATCH^THREAD ([@*procname*] ,[*state*] ,[*event*]);	Returns to frame for immediate dispatch with specified event, after optionally setting current thread procedure and state.
	_RESTORE^THREAD^AND^DISPATCH ([*event*]);	Restores thread procedure and state last pushed, and returns to thread for immediate dispatch with specified event.
	_SAVE^THREAD^AND^DISPATCH ([@*procname*] ,[*state*] ,[*event*]);	Saves current thread procedure and state, and returns to frame for immediate dispatch of new (or same, if none specified) thread procedure in specified state.
	error := _PUSH^THREAD^PROCSTATE ([@*procname*] ,[*state*]); *error* := _POP^THREAD^PROCSTATE;	Saves current thread procedure and state and optionally sets new current thread procedure and state; restores previously pushed thread procedure and state.
	_ST^MIN^THREAD^STATE	Minimum value of user-defined thread state.
	_ST^INITIAL	Thread state value when thread created.

Table 3-1. Summary of I Process Development Library Services (page 5 of 6)

Function	Arguments	Description
CI Communication		
Given that INT .EXT ci-config (_CI^DEF) ...		
Declarations	_CI^ID (*ciid*);	Declares CIID structure where information about an open CI is stored.
	_CI^IDPOINTER (*ciid*);	Declares (extended) pointer to a CIID structure.
Communication	*error* := _OPEN^CI (ci-config, *ciid* ,[*processname*] ,[*nowait-depth*]);	Opens CI for communication.
	error := _SEND^CI (*ciid*, *buffer*, *write-count*, *reply-count* ,[*context-boolean*] ,[*tag*] ,[*timeout*]);	Initiates request for CI communication.
	error := _CANCEL^SEND^CI ([*tag*]);	Cancels outstanding CI communication request.
	error := _CLOSE^CI (*ciid*);	Terminates CI communication.
Information	_LAST^CI^ID	Points to CI that caused last event.
	fserror := _CI^LASTERROR (*ciid*)	File system error of last CI operation.
	replyaddress := _CI^REPLYADDRESS (*ciid*)	Extended address of CI reply buffer.
	replylen := _CI^REPLYLENGTH (*ciid*)	Length of CI reply buffer.
	replytag := _CI^REPLYTAG (*ciid*)	Tag associated with last CI operation.
	filenumber := _CI^FILENUM (*ciid*)	File number of CI involved in most-recently completed communication.
Timeout intervals	CALL _SET^TIMEOUT (*time-interval* ,[*tag*]);	Initiates request for time interval delay.
	error := _CANCEL^TIMEOUT ([*tag*]);	Cancels outstanding timeout.
	_LAST^TIMEOUT^TAG	Accesses tag associated with timeout request.
Command Context		
Defining fixed header portion of command context	_COMMAND^CONTEXT^HEADER ;	Required as part of command context space structure definition to reserve and define input, output, and command context areas.
Accessing command context space	_THREAD^CONTEXT^ADDRESS	Contains extended address of command context space.

Table 3-1. Summary of I Process Development Library Services (page 6 of 6)

Function	Arguments	Description
Frame-defined input/output areas	_INPUT^DEF	Structure template that defines the input area of the command context space.
	_INPUT	Name assigned to the _INPUT^DEF structure by _COMMAND^CONTEXT^HEADER.
	_OUTPUT^DEF	Structure template that defines the output area of the command context space.
	_OUTPUT	Name assigned to the _OUTPUT^DEF structure by _COMMAND^CONTEXT^HEADER.

4 DSNM Command Requirements

Scope of This Section

This section defines the requirements for carrying out DSNM operations. It specifies what information is sent to the command thread and what information must be returned for each command so that the frame can create a DSNM-formatted response to return to the requester.

Command Flow

The typical flow for a DSNM command is from a user to the DSNM command server, and then to one or more I processes. Command responses flow back from the I processes through the command server and then to the user.

The command server is responsible for resolving objects of a command, which means determining their complete set of characteristics and routing them to the correct I process. The I process is responsible for direct communication with the subsystem to carry out commands delivered to it by the command server.

The general command processing flow is as follows:

- The I process carries out the DSNM command for each input list object using subsystem commands. Several subsystem commands may be required to carry out one DSNM command. Input list objects are processed in the order they appear.
- Carrying out a command for an object means executing it for some selection of the input list object and its subordinates in the subsystem hierarchy. The exact set of objects involved is determined by the combination of the hierarchy and state modifiers (see “Object List Modifiers” on page 4-3).
- The I process generates output objects as required by the command. Informational commands other than AGGREGATE return an output object for each object. State-change commands return an output object only in the case of errors. The AGGREGATE command returns summary information rather than information about individual objects.

Command Components

For each DSNM command, the frame places the following components in the command thread context space:

- The action to be performed
- The command modifiers
- A list of objects on which the operation is to be performed

Action to be Performed

The action to be performed is determined by the value in the _INPUT.ACTION field in the command context space. The command thread must be able to translate the following DSNM operations into an equivalent subsystem-specific command or command sequence:

ZDSN^ACTION^ABORT	Brings objects to a nonoperational state, without waiting for outstanding operations to complete.
ZDSN^ACTION^AGGREGATE	Returns a summary of operational status of all objects in the subsystem or under a specified manager process.
ZDSN^ACTION^INFO	Returns configuration information for each object.
ZDSN^ACTION^STATUS	Returns current operational status of each object.
ZDSN^ACTION^START	Brings objects to an operational state.
ZDSN^ACTION^STATISTICS	Returns operational statistics for each object.
ZDSN^ACTION^STOP	Brings objects to a nonoperational state, once outstanding operations are complete.

Command Modifiers

Command modifiers specify whether a command is applied to subordinate objects in the subsystem hierarchy, specify the state of objects to which a command is applied, specify whether error responses are suppressed, and so on. The command modifiers are determined by the values in the _INPUT.MOD structure (ZDSN^MOD^DEF) contained within the command context space. The command modifiers relevant to I processes are listed in Table 4-1.

Table 4-1. Command Modifiers

Modifier	Abbreviation	Modifies/Specifies
Hierarchy	HMOD	Subsystem hierarchy
State	SMOD	State of affected objects
Response	RMOD	Response format
Error	EMOD	Error response format
Action	AMOD	Action of operation

Note: A value of 0 for any modifier indicates that the modifier is omitted.

These modifiers fall into the following categories:

- Object list modifiers, which limit or expand the scope of the original input object list (HMOD and SMOD).
- Response modifiers, which determine the amount and type of information returned for each object in the output object list (RMOD and EMOD).
- The action modifier, which indicates that statistics be reset (AMOD).

Object List Modifiers

Applying the hierarchy and state modifiers to the contents of the original input object list returns a subset or superset of the list. The command thread must be able to resolve the hierarchy and state command modifiers that limit or expand the scope of the input object list.

The Hierarchy Modifier (_INPUT.MOD.Z^HMOD)

The hierarchy modifier (HMOD) controls whether the command is applied to subsystem objects that are subordinate to the object(s) in the input object list. HMOD is valid for all DSNM commands except AGGREGATE. Its values have the following meanings:

ZDSN^HMOD^ALL	For each object on the input object list, apply the command to the object itself and to all subsystem objects subordinate to it. For consistency among subsystems, this should be the default when HMOD is omitted, unless there are overwhelming subsystem reasons for a different default.
ZDSN^HMOD^ONLY	Apply the command to each object on the input object list, but not to subordinate objects.
ZDSN^HMOD^SUBONLY	For each object on the input object list, apply the command only to the subsystem objects that are subordinate to it in the subsystem hierarchy (but not the object itself).

In addition to the HMOD value in the MOD structure supplied with the command (MOD.Z^HMOD), there may also be an HMOD value in the FOBJECT structure associated with each input list object (FOBJ.Z^HMOD). When both are present, the HMOD in the object structure overrides the HMOD command modifier. Table 4-2 summarizes HMOD usage.

Table 4-2. HMOD Usage

FOBJ.Z^HMOD	MOD.Z^HMOD	HMOD Associated With Object
Omitted	Omitted	ZDSN^HMOD^ALL (default)
Omitted	Present	MOD.Z^HMOD
Present	Omitted	FOBJ.Z^HMOD
Present	Present	FOBJ.Z^HMOD

The State Modifier (_INPUT.MOD.Z^SMOD)

The state modifier (SMOD) selects objects for operation according to their current DSNM state. There is no default for SMOD. If omitted, the command should be applied to all objects determined by the HMOD.

SMOD is valid for all DSNM commands except AGGREGATE, INFO and STATISTICS. Its values have the following meanings:

ZDSN^SMOD^UP | ZDSN^SMOD^GREEN	Apply the command only to objects on the object list that are UP (GREEN).
ZDSN^SMOD^NOT^UP | ZDSN^SMOD^NOT^GREEN	Apply the command only to objects on the object list that are DOWN (RED) or PENDING (YELLOW).
ZDSN^SMOD^DOWN | ZDSN^SMOD^RED	Apply the command only to objects on the object list that are DOWN (RED).
ZDSN^SMOD^NOT^DOWN | ZDSN^SMOD^NOT^RED	Apply the command only to the objects on the object list that are UP (GREEN) or PENDING (YELLOW)

Note. ZDSN^SMOD^GREEN and ZDSN^SMOD^UP have the same value and may be used interchangeably. Similarly, NOT^GREEN/NOT^UP, RED/DOWN, and NOT^RED/NOT^DOWN are interchangeable. Externally in DSNM commands, UP, NOT-UP, DOWN, and NOT-DOWN designate these values.

SMOD is applied after HMOD: that is, it can apply to subordinates of an input list object even if the input list object itself does not satisfy the SMOD. For example, a START command specifying a NOT^GREEN SMOD would start subordinates of a GREEN state object in RED and YELLOW states without attempting to start the object itself.

When SMOD is specified, only objects in UP/GREEN, DOWN/RED, and PENDING/YELLOW states are included. So, for example, a STATUS command with a NOT^RED SMOD value should not include UNDEFINED or UNKNOWN objects in the response.

Note. It is the responsibility of the command thread to map the set of states supported by the subsystem into the set of DSNM states (see “Object States” on page 4-7).

Error reporting is independent of SMOD. Unreachable, undefined, or ill-formed objects should be reported as errors according to the EMOD (see the error modifier discussion in “The Error Modifier (_INPUT.MOD.Z^EMOD)” on page 4-6).

Response Modifiers

The response modifiers determine the amount and type of information returned for each object in the output object list.

The Response Modifier (_INPUT.MOD.Z^RMOD)

The response modifier (RMOD) controls the level of detailed information returned for informational commands. RMOD is valid for the STATUS command only; its values have the following meanings:

ZDSN^RMOD^BRIEF	Return the DSNM object state for each object (UP, DOWN, PENDING) and possibly one line of descriptive text (ZDSN^VTY^RESULTTEXT); see “Object States” on page 4-7. This is the default.
ZDSN^RMOD^DETAIL	Return the object state and append as much additional detailed status (ZDSN^VTY^TEXT) as available to each object on the output object list (see individual command descriptions in this section).

Note. SUMMARY-type response modifiers are handled entirely by the I process frame and are never encountered by the command thread itself.

The Error Modifier (_INPUT.MOD.Z^EMOD)

The error modifier (EMOD) controls how much information to return if a subsystem error occurs during command processing. EMOD is valid for all DSNM commands except AGGREGATE; its values have the following meanings:

ZDSN^EMOD^BRIEF	Append a single line of text (ZDSN^VTY^RESULTTEXT) to describe the ZDSN^ERR error code in the Z^RESULT field to members on the output object list that generate an error. This is the default.
ZDSN^EMOD^DETAIL	Append as much available information about the error (ZDSN^VTY^ERRTEXT) to members on the output object list that generate an error, in addition to the EMOD^BRIEF response.
ZDSN^EMOD^SUPPRESS	Suppress the reporting of objects that cause subsystem errors. If the command returns status, configuration, or statistical information, do not create an output object list member for any objects that generate errors.

Any output object that has an error associated with it must contain a ZDSN^ERR code in the Z^RESULT field (see Appendix B, “DSNM Error Codes”).

If ZDSN^EMOD^BRIEF is in effect (the default if EMOD is omitted or has the value 0), one line of result text (ZDSN^VTY^RESULTTEXT) describing the ZDSN^ERR value in the Z^RESULT code field should be appended to the output object (with _APPEND^OUTPUT). The result text must not duplicate the information of the result code. To formulate result text, assume that presentation services will substitute text (listed in Appendix B) for the result code in the error display.

If ZDSN^EMOD^DETAIL is in effect, and if more error information is available from the subsystem than can fit on one line, additional lines of error text (ZDSN^VTY^ERRTEXT) should be appended to the structure to describe the error in detail. (Whether to supply error text depends on the information available from the subsystem; it should be omitted unless there is useful additional information to transmit.)

When the BRIEF response gives all the error information from the subsystem, the DETAIL and BRIEF error responses are identical.

Action Modifiers

ZDSN^AMOD^RESET in the _INPUT.MOD.Z^AMOD field indicates that statistics should be reset after being reported.

The Action Modifier (_INPUT.MOD.Z^AMOD)

The command thread must support the resetting of statistics as indicated by the following action modifier:

ZDSN^AMOD^RESET	Reset subsystem statistics after reporting (valid for STATISTICS command only).

Object States

Subsystem objects can have a number of possible states that are significant within the context of the subsystem. To present uniform status displays of subsystems and their objects, subsystem states are classified into a small set of DSNM states. This set of states can be smaller than the set of subsystem states for an object. Subsystem states are reported as text in the DSNM command response. The command thread must be able to map the states of the subsystem objects to the following DSNM object states:

ZDSN^STATE^UP | ZDSN^STATE^GREEN	Object is in use or available for immediate use.
ZDSN^STATE^DOWN | ZDSN^STATE^RED	The object is unavailable or needs an operator to take action to make it ready.
ZDSN^STATE^PENDING | ZDSN^STATE^YELLOW	The object is neither UP nor DOWN, but is in some intermediate state (such as STARTING). Most subsystems have one or more states to describe an object that is neither ready nor totally deactivated. PENDING corresponds to these subsystem states; an object in this state may require special action to occur or condition to be met.

In addition to the previous states, an object may not be configured, or it may be configured but its state cannot be determined (because a manager is not running or the object is not secured correctly, for instance). For these cases, the following DSNM states are provided:

ZDSN^STATE^UNDEFINED	Object is not defined in the subsystem. An error could have been made, either in configuring the subsystem or entering the object name.
ZDSN^STATE^UNKNOWN	State of the object cannot be determined.
ZDSN^STATE^NULL	Subsystem may have one or more objects that act only to group other objects rather than being functional entities. These objects are represented in DSNM as having a NULL operational state. (This type of object usually doesn't support state-change commands.)

Note. ZDSN^STATE^UP and ZDSN^STATE^GREEN have the same value and may be used interchangeably. Similarly, DOWN/RED and PENDING/YELLOW are interchangeable. Externally in DSNM responses, UP, DOWN, and PENDING are used.

The Input Object List

An object in a DSNM command is a subsystem object name qualified with the subsystem name, the object type, and possibly the object's manager. Depending on the subsystem, a manager may be required, optional, or not allowed.

The input object list consists of input list members, each of which includes a formatted object structure named FOBJ (defined by ZDSN^DDL^FOBJECT^DEF) that describes one object to which the command is to be applied. The command should be applied to objects in the order they appear on the input list.

Subsystems usually have a hierarchy of object types. A DSNM command may specify that the command is applied to only the specified objects, to only their subordinates, or to both.

Each input object list member contains the following FOBJ fields:

Z^HMOD	Is an INT field that contains a hierarchy modifier (HMOD) applying to this object only. If present, it overrides the hierarchy modifier (for this object only) associated with the command as a whole.
Z^SUBSYS	Is a structure (defined by ZDSN^DDL^SUBSYS^DEF) that identifies the subsystem to which the object belongs.
Z^OBJTYPE	Is a structure (defined by ZDSN^DDL^OBJTYPE^DEF) that specifies the subsystem object type of the object.

Z^OBJNAME^ OCCURS	Is an INT field that contains the length of the object name.
Z^OBJNAME	Is a structure (defined by ZDSN^DDL^OBJNAME^DEF) that contains the object name. If the subsystem permits it, the object name may be *, meaning all objects of the object type specified (under the manager specified, if any).
Z^MANAGER^ OCCURS	Is an INT field that contains the length of the manager name. If a manager is not present, Z^MANAGER^OCURS is 0.
Z^MANAGER	Is a structure (defined by ZDSN^DDL^MANAGER^DEF) that contains the name of the manager process, if any.

Execution Objects

The input list objects and the hierarchy and state modifiers determine the final set of objects to which the command is to be applied.

Applying Object List Modifiers

The final list of objects to which a command is eventually applied results from the application of the hierarchy and state modifiers to the members of the input object list in the following order:

1. Apply the Z^HMOD value associated with the command (*context-area*._INPUT.MOD.Z^HMOD).
2. Apply the Z^HMOD value, if it exists, within the individual formatted object structures (*context-area*.*output-list-member*.FOBJ.Z^HMOD).
3. Apply the Z^SMOD value associated with the command (*context-area*._INPUT.MOD.Z^SMOD).

Use the list-processing library services described in “Processing a List” later in this section to manipulate the original input list and to create intermediate lists.

The User Area: Intermediate Lists

A context space is allocated to each thread when created, and persists until the thread terminates. The context space contains a fixed header area reserved for use by the frame, followed by a user-defined area that can be used as workspace to manipulate intermediate object lists. See “Command Context Space” on page 3-15 for information on accessing the user-defined area of the context space.

The Output Object List

The output object list is built by the command thread. Each member includes a formatted object structure named FOBJ (defined by ZDSN^DDL^FOBJECT^DEF) that describes one object to which the command was applied.

Depending on the command, and the hierarchy, state, and error modifiers, a single input list object may produce many output objects (which may or may not include the original input list object), or no output objects at all.

With minor exceptions (see detailed command descriptions in this section), the command thread must always fill the following fields in each output object:

Z^RESULT	Is an INT field containing the result code for the output object. It can be one of the ZDSN^ERR values (see Appendix B), a ZDSN^STATE value (for the STATUS command), or null (0 value). Except for STATUS command responses, ZDSN^ERR^NOERR (0 value) is used in all responses when no error occurs.
Z^SUBSYS	Is a structure (defined by ZDSN^DDL^SUBSYS^DEF) that identifies the subsystem to which the object belongs.
Z^OBJTYPE	Is a structure (defined by ZDSN^DDL^OBJTYPE^DEF) that specifies the subsystem object type of the object.
Z^OBJNAME	Is a structure (defined by ZDSN^DDL^OBJNAME^DEF) that contains the object name, terminated with a blank or null.
Z^MANAGER	Is a structure (defined by ZDSN^DDL^MANAGER^DEF) that contains the name of the manager process (if any), terminated with a blank or null. If there is no manager, this field should be blank or null (0 value).

Output Object Variable-Length Items

Depending on specific command details, one or more of the following variable-length items can be appended to an output object (with _APPEND^OUTPUT):

ZDSN^VTY^RESULTTEXT	Interprets the Z^RESULT code, either describing the subsystem state of the object (for STATUS commands) or providing error information.
ZDSN^VTY^TEXT	Is the response text for STATUS (DETAIL), INFO, or STATISTICS commands.
ZDSN^VTY^ERRTEXT	Is detailed error text.
ZDSN^VTY^COUNTERS	Is the state summary counters for the AGGREGATE command.

The maximum length of a text line (RESULTTEXT, TEXT, or ERRTEXT items) that may be appended is 75 characters (ZDSN^MAX^TEXT).

Command Requirements

The DSNM commands are often needed in daily operations and can be applied to widely diverse objects. For the most part, subsystems support commands equivalent to the DSNM commands for their objects. There are two categories of DSNM commands: informational commands and state-change commands:

- Informational commands (AGGREGATE, INFO, INQUIRE, STATISTICS, and STATUS) require that the output list contain a formatted object for each object on the input list.
- State-change commands (ABORT, START, STOP, and UPDATE) require only exception output.

The remainder of this section details the valid command modifiers and response requirements for DSNM commands, except for the INQUIRE and UPDATE commands: the I processes and the frame are not involved in these command operations.

The ABORT Command

ABORT issues the subsystem command(s) that immediately bring objects to a nonoperational state without waiting for outstanding operations to finish. (A nonoperational state is the subsystem state equivalent to the DSNM state ZDSN^STATE^DOWN or ZDSN^STATE^RED.)

Note. If the subsystem makes no distinction between stopping objects gracefully and otherwise, the DSNM STOP and ABORT commands perform the same operation.

Valid Modifiers

HMOD, EMOD, and SMOD.

RMOD does not apply and should be ignored if present.

Output Object Requirements

Commands to abort subsystem objects should be issued for objects obtained by applying the hierarchy (HMOD) and state (SMOD) modifiers to each input list object.

ABORT must be performed on objects in the order they appear in the input list.

Build an output object structure for only those objects that cannot be aborted, consistent with the EMOD value (see the error modifier discussion in the “The Error Modifier (_INPUT.MOD.Z^EMOD)” on page 4-6).

Objects aborted successfully do not generate a response. ZDSN^EMOD^SUPPRESS causes all objects to be omitted from the response.

Note. Many subsystems produce a warning if an ABORT operation is issued for an object that is already stopped. Such a warning should be ignored; do not report it as an error.

The AGGREGATE Command

AGGREGATE issues the subsystem command(s) that return the current operational status of objects. This command summarizes the operational status of all objects in a subsystem. If the subsystem employs a manager, the summary is for all objects controlled by the manager. If there is no manager, Z^MANAGER^OCCURS is 0.

For the input list object, only the fields Z^SUBSYS, Z^MANAGER^OCCURS and Z^MANAGER are relevant. Object type, object name, and HMOD are irrelevant and should be ignored.

Valid Modifiers

None; any modifiers present should be ignored.

Output Object Requirements

For each input list object, return one output object for each object type in the subsystem, designated as follows:

Z^SUBSYS	Subsystem
Z^OBJTYPE	Object type
Z^OBJNAME	Blank
Z^MANAGER	Manager, if any; blank otherwise
Z^RESULT	Null (0 value)

Append a type ZDSN^VTY^COUNTERS counters structure (described by ZDSN^DDL^COUNTERS^DEF) containing the number of objects of Z^OBJTYPE in each DSNM state. The relevant counters structure fields are:

INT(32) Z^GREEN;

INT(32) Z^UP = Z^GREEN;

INT(32) Z^RED;

INT(32) Z^DOWN = Z^RED;

INT(32) Z^YELLOW;

INT(32) Z^PENDING = Z^YELLOW;

INT(32) Z^UNDEFINED;

INT(32) Z^INERROR;

Count objects in ZDSN^STATE^NULL in the Z^GREEN counter, which here is interpreted as “exists.”

Accumulate objects in both ZDSN^STATE^UNDEFINED and ZDSN^STATE^UNKNOWN in the Z^UNDEFINED counter.

If an error occurs such that the subsystem or a manager cannot be reached to carry out the AGGREGATE command, return one output object for that input object in the ZDSN^EMOD^BRIEF format. If necessary, append one line of ZDSN^VTY^RESULTTEXT, further describing the error. Designate the object structure fields as follows:

Z^SUBSYS	Subsystem
Z^OBJTYPE	Blank
Z^OBJNAME	Blank
Z^MANAGER	Manager, if any; blank otherwise
Z^RESULT	ZDSN^ERR code describing the error

If necessary, append one line of ZDSN^VTY^RESULTTEXT, further describing the error.

The INFO Command

INFO issues the subsystem command(s) that return configuration information for objects.

Valid Modifiers

HMOD and EMOD.

SMOD is not supported for the INFO command and should be ignored if present.

Output Object Requirements

Return one output object for each hierarchically derived input list object (unless Z^EMOD = ZDSN^EMOD^SUPPRESS, which suppresses the reporting of objects that cause subsystem errors).

Return Z^RESULT null (0 value) unless an error occurs. Do not append ZDSN^VTY^RESULTTEXT except to further interpret a returned error value.

Append ZDSN^VTY^TEXT lines to report all normal configuration information for each object.

The START Command

START issues the subsystem command(s) to bring objects to an operational state (the subsystem state equivalent to the DSNM state ZDSN^STATE^UP or ZDSN^STATE^GREEN).

Valid Modifiers

HMOD, EMOD, SMOD.

RMOD does not apply and should be ignored if present.

Output Object Requirements

Commands to start subsystem objects should be issued for objects obtained by applying the hierarchy (HMOD) and state (SMOD) modifiers to each input list object.

START must be performed on objects in the order they appear in the input list.

Build an output object structure for only those objects that cannot be aborted, consistent with the EMOD value (see the error modifier discussion in “The Error Modifier (_INPUT.MOD.Z^EMOD)” on page 4-6).

Objects started successfully do not generate a response. ZDSN^EMOD^SUPPRESS causes all objects to be omitted from the response.

Note. Many subsystems produce a warning if a START operation is issued for an object that is already started. Such a warning should be ignored; do not report it as an error.

The STATISTICS Command

STATISTICS issues the subsystem command(s) that return operational statistics for objects.

Valid Modifiers

HMOD, EMOD, AMOD.

AMOD = ZDSN^AMOD^RESET (meaning statistics counters are to be reset after being reported).

SMOD is not supported and should be ignored if present.

Output Object Requirements

Return one output object for each hierarchically derived input list object (unless Z^EMOD = ZDSN^EMOD^SUPPRESS, which suppresses the reporting of objects that cause subsystem errors).

Return Z^RESULT null (0 value) unless an error occurs. Do not append ZDSN^VTY^RESULTTEXT except to further interpret a returned error value.

Append ZDSN^VTY^TEXT lines to report all normal statistical information for each object.

If ZDSN^AMOD^RESET is in effect, reset object statistics after reporting them in the response.

The STATUS Command

STATUS issues the subsystem command(s) that return the current operational status of objects. One output object should be returned for each subsystem object obtained by applying the hierarchy (HMOD) and state (SMOD) modifiers to each input list object.

Valid Modifiers

HMOD, EMOD, SMOD, RMOD.

Output Object Requirements

Return one output object for each hierarchically derived input list object (unless Z^EMOD = ZDSN^EMOD^SUPPRESS, which suppresses the reporting of objects that cause subsystem errors).

The command thread must determine the subsystem state of each object and translate it into a DSNM state. An object may have more than one subsystem state attribute, which is relevant to the operational state of the object, and which is a factor in determining the DSNM state. The command thread must translate subsystem-derived information into a DSNM state.

Return the DSNM state of the object in the Z^RESULT field.

If the subsystem state(s) used to determine the DSNM state add relevant operational information to the Z^RESULT code, report it by appending one line of ZDSN^VTY^RESULTTEXT. (Result text should not repeat the DSNM state itself; it should provide additional information). PENDING states most often require additional interpretation.

If ZDSN^RMOD^DETAIL is in effect, append additional ZDSN^VTY^TEXT entries, providing all operational information available from the subsystem. These additional text lines should augment, not replace, the ZDSN^VTY^RESULTTEXT information (see example below).

Example

The following command requests brief status information (the default) for a SNAX line and its subordinate PUs and LUs:

```
STATUS SNAX LINE \WYJ.$STLR
       .
       .
SNAX LU      \WYJ.$STLR.#TLR1 Pending, Stopping
       .
       .
```

In the resulting display, `Stopping` is the appended ZDSN^VTY^RESULTTEXT, clarifying the PENDING state returned in Z^RESULT.

The following command line requests detailed status information:

```
STATUS SNAX LINE \WYJ.$STLR, DETAIL
```

The resulting display might include something like the following. The text in bold is ZDSN^VTY^TEXT appended to the output object.

```
        .
        .
SNAX LU      \WYJ.$STLR.#TLR1 Pending, Stopping
   Lu State: Daclu Request Pending, Not in Session
   Session State: Not in Session
   Open State: Opens Forbidden
   Session Id: 2
   Sw Line Name : $STLR”
```

The STOP Command

STOP issues the subsystem command(s) that bring objects to a nonoperational state (the subsystem state equivalent to the DSNM state ZDSN^STATE^DOWN or ZDSN^STATE^RED). Objects should be brought down gracefully if the subsystem supports it, allowing current operations to terminate normally.

Note. If the subsystem makes no distinction between stopping objects gracefully and otherwise, the DSNM STOP and ABORT commands perform the same operation.

Valid Modifiers

HMOD, EMOD, SMOD.

RMOD does not apply and should be ignored if present.

Output Object Requirements

Commands to stop subsystem objects should be issued for objects obtained by applying the hierarchy (HMOD) and state (SMOD) modifiers to each input list object.

STOP must be performed on objects in the order they appear in the input list.

Build an output object structure for only those objects that cannot be aborted, consistent with the EMOD value (see the error modifier discussion in “The Error Modifier (_INPUT.MOD.Z^EMOD)” on page 4-6).

Objects stopped successfully do not generate a response. ZDSN^EMOD^SUPPRESS causes all objects to be omitted from the response.

Note. Many subsystems produce a warning if a STOP operation is issued for an object that is already stopped. Such a warning should be ignored; do not report it as an error.

---

5 DSNM Process Startup Functions

Scope of This Section

DSNM processes call a startup procedure as the first step in the main procedure. Typically, the startup procedure reads parameters via library calls, and then continues with other process-specific initialization.

Processes retrieve process parameters and configuration parameters from either the startup message or the DSNMCONF file, depending on how the system is configured and how the parameter-retrieval procedures are called. Processes retrieve subsystem and CI configuration information from the DSNMCONF file.

The structure and content of the DSNMCONF file is discussed in Section 6, "Configuring a New Subsystem Into DSNM." This section describes:

- The startup message
- The ways in which startup messages and configuration files are searched
- How the structures into which processes retrieve startup and configuration parameters are declared and defined
- The intended usage and syntax of the library procedures that retrieve the following information into predefined structures for use by the frame and the command thread:
  - Process parameters and configuration parameters from the startup message and the DSNMCONF file
  - Subsystem configuration information
  - CI configuration information

DSNM Process Startup Message

The format of the parameter portion of the RUN command for DSNM processes appears in bold next. These are the parameters that are sent to the new process in the startup message.

```
[ RUN ] program-file [ / run-option [ , run-option ] ... / ]
[ process-parameter [, process-parameter ] ...]
[; DSNM-config-parameter [, DSNM-config-parameter ] ...]
```

Process parameters are specific to the particular process itself. These parameters are accepted by the process, regardless of the STARTUP [PARAMS] value in the $SYSTEM.SYSTEM.DSNM file (see Section 6, "Configuring a New Subsystem Into DSNM").

DSNM configuration parameters are ignored unless STARTUP [PARAMS] is YES in the environment (see "DSNM Configuration Parameters" on page 5-3). If you specify DSNM configuration parameters but no process parameters, the configuration parameter list must begin with a semicolon (;).

Process Parameters

The standard DSNM process parameters accepted from the startup message are discussed next. (TESTMODE, CONFIG, STARTUP, and DEBUG are used for development testing only.)

`DSNM` *`env-name`*

specifies the $SYSTEM.SYSTEM.DSNM section name to be used by this process. The process uses the environment defined in ?SECTION *`env-name`* in $SYSTEM.SYSTEM.DSNM. If both the DSNM and CONFIG parameters are omitted, the unnamed section (blank) of $SYSTEM.SYSTEM.DSNM is used.

If $SYSTEM.SYSTEM.DSNM exists, it must contain a section named *`env-name`* or a fatal error is reported.

`COMPONENT` *`component-name`*

specifies the process component name. This value is used for retrieval of parameter values from the DSNMCONF file.

For I processes, COMPONENT is usually the name of the subsystem the I process handles. For I processes that handle more than one subsystem (such as the SCP I process), the component name is an arbitrary name chosen by the developer of the process (for example, COMM is the component name for the SCP I process).

`MYSYSTEM` *`system-name`*

specifies the acting home system, if the process can act as if a remote system were its home system.

`TESTMODE` *`num`*

any nonzero value specifies that the process is running in test mode; 0, the default, specifies production mode. This parameter is valid only if the process is compiled in test mode; you receive a fatal error if the process is compiled in production mode.

Test mode forces the STARTUP [PARAMS] value in $SYSTEM.SYSTEM.DSNM to default to YES, and enables processing of the CONFIG, STARTUP, and DEBUG process parameters.

Note. To compile in test mode, set toggle 1 during compilation (SETTOG 1). Recompile all programs without SETTOG 1 before placing them in production.

`CONFIG` *`filename`* [*`filename`*] `...`

specifies a DSNMCONF search list of up to three configuration files, which overrides the DSNMCONF file pointed to in $SYSTEM.SYSTEM.DSNM.

CONFIG allows you to specify multiple configuration files for testing purposes only. This allows you to maintain your subsystem, CI, and I process configuration records separate from your installation's site-specific production environment configuration. CONFIG values are ignored if TESTMODE is not enabled.

To start an I process for testing with DSNMCom (I process test utility), using site-specific configuration parameters in $DSNM.NETW.DSNMCONF and I-process-specific configuration parameters in $DSNM.IDEV.DSNMIDEV, use the TESTMODE and CONFIG process parameters in an explicit RUN command. For example:

```
RUN $DSNM.IDEV.SPIFI/NAME $SPFI,NOWAIT/TESTMODE 1, &
     CONFIG $DSNM.NETW.DSNMCONF $DSNM.IDEV.DSNMIDEV
```

`STARTUP { YES | NO }`

specifies whether DSNM configuration parameters from the startup message are allowed to override parameters stored in the DSNMCONF file(s). (This is the same as the STARTUP [PARAMS] value in the $SYSTEM.SYSTEM.DSNM file.) The STARTUP value is ignored if TESTMODE is not enabled. DSNM configuration parameters are discussed in the next subsection.

`DEBUG` *`num`*

enables DEBUG calls as specified by *`num`*. DEBUG *`num`* sets the Z^DEBUG^LEVEL field in the _PROCESS^PARAMS structure to *`num`* (see “Accessing Standard Process Parameters: _PROCESS^PARAMS” on page 5-8). Its purpose is to call DEBUG under various externally specified circumstances unique to the particular process. The process developer decides what each value of *`num`* means; there are no external standards for it. This parameter is ignored if TESTMODE is not enabled.

DSNM Configuration Parameters

DSNM configuration parameters are site-specific parameters such as SWAPVOL; these parameters are described in the *Distributed Systems Management Solutions (DSMS) System Management Guide*.

`DSNM-config-parameter` may be accepted from the startup message if either:

- STARTUP [PARAMS] is YES in the designated section of $SYSTEM.SYSTEM.DSNM (in this case the *`DSNM-config-parameter`* overrides or supplements the value of the corresponding parameter in the DSNMCONF file).
- The process is running in test mode (compiled with SETTOG 1) and the process parameters TESTMODE 1 (or any nonzero value) and STARTUP YES are specified.

DSNM configuration parameters are separated from standard process parameters by a semicolon (;).

Note. If STARTUP [PARAMS] is NO in the designated section of $SYSTEM.SYSTEM.DSNM, DSNM configuration parameters in the startup message are ignored, and no error is reported.

Parameter Types and Search Criteria

The parameter retrieval library functions retrieve parameter values from the startup message or the DSNMCONF file, according to the search criteria specified in the arguments passed to the procedures. (See the syntax descriptions of the procedures in “Parameter Retrieval Library Services” on page 5-6.) Such library functions return either the first value set obtained or the union of value sets obtained, depending on whether the parameter is local or global.

A value set is all the records retrieved from a single configuration file (or from the startup message) by a generic key search that excludes the SEQUENCE field.

If STARTUP [PARAMS] is set to YES, and if at least one instance of the named parameter is found in the startup message, the startup message values determine the value set. If not, the configuration file is searched for the first key. The first successful search determines the value set. If no search is successful with the first key, the procedure is repeated with the second key, and so on.

Note. In the following discussion on local and global parameters, *mysystem* refers to either the local system, or, if the process accepts the MYSYSTEM process parameter, the system that acts as the home system. Also, if the process has no component name (“*class component*” is identical to “*class* blank”), only one search is performed.

Local Parameters and Search Patterns

Local parameters consist of a single value (for example, SWAPVOL) obtained from one source: a DSNMCONF file or the startup message.

Local Component Parameters

A local component parameter is a parameter specific to this component and class. The first value set found by the following generic key search is returned:

mysystem	DSNM	*class*	*component*	*parameter*
	DSNM	*class*	*component*	*parameter*

Local Class Parameters

A local class parameter is a parameter specific to this class. If *component* is blank, it is specific to the class as a whole. The first value set found by the following generic key search is returned:

mysystem	DSNM	*class*	*component*	*parameter*
mysystem	DSNM	*class*	blank	*parameter*
blank	DSNM	*class*	*component*	*parameter*
blank	DSNM	*class*	blank	*parameter*

Local General Parameters

A local general parameter is any instance of this parameter. It may be for this component, for the class as a whole (if *component* is blank), or for any class (if both *class* and *component* are blank). The first value set found by the following generic key search is returned:

mysystem	DSNM	*class*	*component*	*parameter*
mysystem	DSNM	*class*	blank	*parameter*
mysystem	DSNM	blank	blank	*parameter*
blank	DSNM	*class*	*component*	*parameter*
blank	DSNM	*class*	blank	*parameter*
blank	DSNM	blank	blank	*parameter*

Global Parameters and Search Patterns

Global parameters consist of multiple value sets: the union of sets of values from all sources in which instances of the parameter are found. The value set for a global parameter typically contains multiple values (for example, command server SYSTEM parameters).

Global Component Parameters

A global component parameter is all instances of this parameter specific to this component and class. All value sets found by the following generic key searches are returned:

mysystem	DSNM	*class*	*component*	*parameter*
	DSNM	*class*	*component*	*parameter*

Global Class Parameters

A global class parameter is all instances of this parameter specific to this class. If *component* is blank, it is specific to the class as a whole. All value sets found by the following generic key searches are returned:

mysystem	DSNM	*class*	blank	*parameter*
mysystem	DSNM	*class*	*component*	*parameter*
blank	DSNM	*class*	*component*	*parameter*
blank	DSNM	*class*	blank	*parameter*

Global General Parameters

A global general parameter is all instances of this parameter. It may be for this component, for the class as a whole (if *component* is blank), or for any class (if both *class* and *component* are blank). All value sets found by the following generic key searches are returned:

mysystem	DSNM	*class*	*component*	*parameter*
mysystem	DSNM	*class*	blank	*parameter*
mysystem	DSNM	blank	blank	*parameter*
blank	DSNM	*class*	*component*	*parameter*
blank	DSNM	*class*	blank	*parameter*
blank	DSNM	blank	blank	*parameter*

Parameter Retrieval Library Services

In the context of an I process, the frame performs the following steps in its startup procedure:

1. The frame calls _STARTUP^MODE.

   _STARTUP^MODE is a user-written procedure that provides the frame with:

   - The COMPONENT name for configuration parameter retrieval searches. The component name is usually the name of the subsystem the I process handles. For I processes that handle more than one subsystem (such as the SCP I process) the component name is an arbitrary name chosen by the developer of the process (for example, COMM is the component name for the SCP I process).
   - A value indicating whether the I process is running in test mode or production mode.
   - A value indicating whether a COMPONENT value in the startup message should be accepted as an overriding value.

2. The frame then calls a frame procedure which:

   - Opens $RECEIVE.
   - Reads the startup message.
   - Reads $SYSTEM.SYSTEM.DSNM (if indicated).
   - Opens the appropriate configuration file.

- Uses the COMPONENT value passed from _STARTUP^MODE (or the startup message, if indicated) to retrieve standard process parameters and configuration parameters into predefined structures.
- Supplies appropriate defaults.

Note. The command thread also has access to the information retrieved by the frame; see the next two subsections for information on accessing _PROCESS^PARAMS and _DSNMCONF^PARAMS parameters.

---

3. The frame calls the _STARTUP procedure.

   _STARTUP is another user-written procedure that supplies the lengths of the user context area and the input list members. It also retrieves and places subsystem and CI configuration parameters into predefined structures for use by the frame.

   The following procedures must be called in your I process _STARTUP procedure:

   - _ADD^SUBSYS: fills in a predefined structure with subsystem configuration parameters for the subsystem(s) the I process handles. The frame uses this information when it gets a command for that subsystem.
   - _ADD^CI: fills in a predefined structure with CI configuration parameters for the CI class with which your I process communicates.

   In addition, your I process _STARTUP procedure may call the following procedures:

   - _GET^PROCESS^PARAM: retrieves process parameter values that are not part of the standard set retrieved by the frame and stored in _PROCESS^PARAMS.
   - _GET^PARAM: retrieves configuration parameter values that are not part of the standard set retrieved by the frame and stored in _DSNMCONF^PARAMS.

4. The frame then terminates startup processing, closes the open DSNMCONF file, and frees resources allocated on behalf of the configuration library procedures.

Accessing Standard Process Parameters: _PROCESS^PARAMS

As part of its startup processing, the frame retrieves values for these parameters and fills in a structure declared by _PROCESS^PARAMS. The command thread may then access these values for its own use.

The structure declared by _PROCESS^PARAMS is defined as follows:

```
DEFINITION ZDSN-DDL-PROCESS-PARAMS.
  02 Z-CLASS-OCCURS         TYPE ZSPI-DDL-UINT.
  02 Z-CLASS                TYPE ZDSN-DDL-CLASS.
  02 Z-COMPONENT-OCCURS     TYPE ZSPI-DDL-UINT.
  02 Z-COMPONENT            TYPE ZDSN-DDL-COMPONENT.
  02 Z-MYSYSTEM-OCCURS      TYPE ZSPI-DDL-UINT.
  02 Z-MYSYSTEM             TYPE ZDSN-DDL-SYSTEM.
  02 Z-MYREALSYSTEM-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-MYREALSYSTEM         TYPE ZDSN-DDL-SYSTEM.
  02 Z-MYPROCESS-OCCURS     TYPE ZSPI-DDL-UINT.
  02 Z-MYPROCESS            TYPE ZDSN-DDL-PNAME.
  02 Z-TESTMODE             TYPE ZSPI-DDL-INT.
  02 Z-DEBUG-LEVEL          TYPE ZSPI-DDL-ENUM.
  02 Z-SECTION-NAME-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-SECTION-NAME         TYPE ZDSN-DDL-PARAMNAME.
END
```

Accessing Standard Configuration Parameters: _DSNMCONF^PARAMS

In addition to the startup message process parameters, a set of configuration parameters also applies to many DSNM processes. The standard configuration parameters are:

DSNM-MANAGER
EMS-COLLECTOR
MAXOPENERS
OBJECT-DB
OBJECT-DB-INTERFACE
OBJECT-MONITOR
SEGEXT
SEGPAGES
SWAPVOL

As part of its startup processing, the frame retrieves values for these parameters and fills in a structure declared by __DSNMCONF^PARAMS. The command thread may then access these values for its own use.

The structure declared by _DSNMCONF^PARAMS is defined as follows:

```
DEFINITION ZDSN-DDL-DSNMCONF-PARAMS.
  02 Z-DSNM-MANAGER-OCCURS   TYPE ZSPI-DDL-UINT.
  02 Z-DSNM-MANAGER          TYPE ZDSN-DDL-MANAGER.
  02 Z-SWAPVOL-OCCURS        TYPE ZSPI-DDL-UINT.
  02 Z-SWAPVOL               TYPE ZDSN-DDL-OBJNAME.
  02 Z-SEGPAGES              TYPE ZSPI-DDL-INT2.
  02 Z-SEGEXT.
    03 Z-PRIMARY             TYPE ZSPI-DDL-INT.
    03 Z-SECONDARY           TYPE ZSPI-DDL-INT.
  02 Z-OBJECT-DB-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-OBJECT-DB             TYPE ZDSN-DDL-OBJNAME.
  02 Z-OBJECT-MONITOR-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-OBJECT-MONITOR        TYPE ZDSN-DDL-PNAME.
  02 Z-OBJECT-DB-INTERFACE-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-OBJECT-DB-INTERFACE   TYPE ZDSN-DDL-PNAME.
  02 Z-MAX-OPENERS           TYPE ZSPI-DDL-INT.
  02 Z-EMS-COLLECTOR-OCCURS  TYPE ZSPI-DDL-UINT.
  02 Z-EMS-COLLECTOR         TYPE ZDSN-DDL-PNAME.
  02 Z-SECPARAMS             TYPE ZSPI-DDL-UINT.
END
```

Note. Values configured for Z-SEGEXT are ignored by all DSNM processes supplied by Tandem.

Retrieving Non-Standard Process Parameters: _GET^PROCESS^PARAM

You can call _GET^PROCESS^PARAM in your _STARTUP procedure to retrieve process startup parameter values that are not part of the standard set stored in the _PROCESS^PARAMS structure.

```
error := _GET^PROCESS^PARAM ( paramname
                            , paramvalue:maxlen
                            ,[ len ] );
```

error output

is a ZDSN or NonStop Kernel error. (FEEOF means there are no more parameters with this name.)

paramname input

STRING .EXT ! ZDSN^DDL^PARAMNAME^DEF !

is the parameter name, left-justified, blank-filled.

paramvalue output

STRING .EXT

is the parameter value.

maxlen input

INT

is the maximum number of bytes that can be returned in *paramvalue*.

len output

INT

is the number of bytes returned in *paramvalue*.

Retrieving Nonstandard Configuration Parameters: _GET^PARAM

You can call _GET^PARAM in your _STARTUP procedure to retrieve configuration parameter values that are not part of the standard set stored in the _DSNMCONF^PARAM structure.

```
error := _GET^PARAM ( paramscope
                    , paramtype
                    ,[ subsys ]
                    ,[ class ]
                    ,[ component ]
                    , paramname
                    , paramvalue:maxlen
                    ,[ len ]
                    ,[ error-filename ] );
```

error output

is a ZDSN or NonStop Kernel error. (FEEOF means there are no more parameters with this name.)

paramscope input

INT

indicates whether the parameter is local or global:

_LOCAL^PARAM	Local parameters consist of a single value (for example, SWAPVOL) obtained from one source—a DSNMCONF file or the startup message.
_GLOBAL^PARAM	Global parameters consist of multiple values (for example, command server SYSTEM parameters) from all sources in which instances of the parameter are found.

paramtype input

INT

indicates how restrictive the search criteria is:

_COMPONENT^PARAM	Component parameters are instances of a parameter specific to this component and class.
_CLASS^PARAM	Class parameters are instances of a parameter specific to this class. If *component* is blank, it is specific to the class as a whole.
_GENERAL^PARAM	General parameters are any instance of this parameter. It may be for this component, for the class as a whole (if *component* is blank), or for any class (if both *class* and *component* are blank).

subsys input

STRING .EXT ! ZDSN^DDL^SUBSYS^DEF !

is the name of the subsystem whose associated parameter values are to be retrieved. A blank subsystem name (all spaces) is valid; the default is “DSNM ”.

class input

STRING .EXT ! ZDSN^DDL^CLASS^DEF !

is the name of the class whose associated parameter values are to be retrieved. A blank class name (all spaces) is valid; if omitted, the caller’s class name is used.

component input

STRING .EXT ! ZDSN^DDL^COMPONENT^DEF !

is the name of the component whose associated parameter values are to be retrieved. A blank component name (all spaces) is valid; if omitted, the caller’s component name (specified by the COMPONENT parameter in the _STARTUP^MODE procedure or obtained from the process startup message) is used.

paramname input

STRING .EXT ! ZDSN^DDL^PARAMNAME^DEF !

is the name of the parameter, left-justified, blank-filled, whose value you want returned.

paramvalue output

STRING .EXT

contains the parameter value if error = 0; otherwise undefined.

maxlen input

INT

the maximum length, in bytes, that can be returned in *paramvalue*.

len output

INT

the actual length, in bytes, of the value returned in *paramvalue*. If *len* < *maxlen*, the remainder of *paramvalue* is blank-filled.

error-filename output

STRING .EXT ! ZDSN^DDL^OBJNAME^DEF !

is the name of the configuration file associated with the returned *error* value.

Retrieving Subsystem Configuration Parameters

For each subsystem it handles, the I process must declare an extended pointer to a subsystem configuration structure defined by _SUBSYS^DEF in its global definitions. Then, as part of its _STARTUP procedure, it must call _ADD^SUBSYS to retrieve subsystem configuration information, and to allocate the memory for, fill in, and return the address of each _SUBSYS^DEF-declared structure. The frame uses information from this structure when it gets a command for that subsystem.

Retrieving CI Configuration Parameters

For each subsystem manager process (CI) it communicates with, the I process must declare an extended pointer to a CI configuration structure defined by _CI^DEF in its global definitions. Then, as part of its _STARTUP procedure, it must call _ADD^CI to retrieve CI configuration information, and to allocate the memory for, fill in, and return the address of each _CI^DEF-declared structure. The frame uses information from this structure when it gets a request from the thread to open a CI for communication.

6

Conf

DSNM

Scope of This Section

This section contains the steps necessary to configure a subsystem and its associated I process into DSNM. More configuration and management details, including an extended list of the DSNM process class configuration parameters, can be found in the *Distributed Systems Management Solutions (DSMS) System Management Guide*.

As an I process developer, you must provide the person responsible for installing and managing DSNM at your site with configuration information specific to your I process, the subsystem(s) it manages, and the subsystem manager process (CI) with which it communicates.

Note. For testing purposes, it is convenient to maintain all of your subsystem, CI, and I process configuration records in a separate configuration file. If you compile in test mode, you can then run your I process with the TESTMODE and CONFIG process parameters to specify this file.

New and Changed DSNM Configuration Information

DSNM now runs "out-of-the-box," which means DSNM can execute without customization after the Install REPSUBSYS phase. DSMS processes use internal default values that previously had to be specified in various configuration files. All user-supplied configuration files are optional, and the post-Install function (DINSTALL) has been eliminated.

Note. Verify that your system is updated to the C30 DSMS release before installing C31 or a later release. The C30 DSMS release contained major changes in the handling of DSMS process and file names.

Note the following important highlights of the DSNM customization changes:

- By default, DSMS now operates with most files in a single subvolume (the ISV or a copy of the ISV) rather than being distributed among several subvolumes as in previous releases.
- DSMS now creates the object database if it does not exist.
- $SYSTEM.SYSTEM.ZDSNCONF is a new file (since the C31 release) and is installed by the REPSUBSYS phase of the Install program. This file is a key-sequenced file and contains DSNM configuration parameters supplied by Tandem.

  $SYSTEM.SYSTEM.ZDSNCONF may change at each release, but you should not alter it yourself. If you need to override particular parameters for your installation, refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide*.
- The $SYSTEM.SYSTEM.DSNM file is no longer a required file.

- The following are changes to the processing of the DSNM configuration file:
  - If no DSNM configuration file is specified, each DSMS process uses the file named DSNMCONF, located on the same subvolume as the object file of the process.
  - If a nonexistent DSNM configuration file is specified (explicitly or by default), the DSNM processes behave as if the file is present, but empty.
  - All DSNM configuration file parameters for DSMS processes supplied by Tandem now have internal default values. Additionally, a number of DSNM configuration file parameters have been added or altered for C31 and subsequent releases. Refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide* for the complete description of a DSNM configuration parameters.
- The following are changes to the processing of the $SYSTEM.SYSTEM.DSNM file:
  - If $SYSTEM.SYSTEM.DSNM does not exist or is unreadable, each DSMS process uses the default DSNM configuration file (DSNMCONF) and the default value STARTUP PARAMS YES.
  - The CONFIG parameter in $SYSTEM.SYSTEM.DSNM is now optional. If present, it may contain a search list of one to three files, all of which must be in the format of a DSNM configuration file. (The specified files are searched for parameters in the order listed.) If CONFIG is absent, the default DSNM configuration file is used. In either case, $SYSTEM.SYSTEM.ZDSNCONF is searched for parameters after all other DSNM configuration files. Refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide* for details on the search order.
  - Comments are now allowed in the $SYSTEM.SYSTEM.DSNM file.

The $SYSTEM.SYSTEM.DSNM File

$SYSTEM.SYSTEM.DSNM is an edit file that all DSNM processes in a production environment read as part of their startup function. It points to the DSNMCONF file from which configuration parameters are retrieved, and specifies whether parameter values in the DSNMCONF file can be overridden by parameter values from the startup message (startup message parameters are discussed in Section 5, “DSNM Process Startup Functions”).

Note. In a test environment, you specify a DSNMCONF file with the CONFIG process parameter in your I process RUN command.

The format of $SYSTEM.SYSTEM.DSNM is as follows:

```
[ ?SECTION ]
  [ CONFIG [ FILE ] filename  [ filename [ filename ] ] ]
  [ STARTUP [ PARAMS ] { YES | NO } ]

[ ?SECTION  env-name
  [ CONFIG [ FILE ] filename  [ filename [ filename ] ] ]
  [ STARTUP [ PARAMS ] { YES | NO } ] ]
       .
       .
       .
```

?SECTION *env-name*

names the DSNM environment. Each section in $SYSTEM.SYSTEM.DSNM defines a separate environment.

The first section in $SYSTEM.SYSTEM.DSNM is always the unnamed section. The unnamed section defines the default environment. The ?SECTION statement is optional for the unnamed section. An unnamed (blank) ?SECTION statement that is not the first section in the file is ignored. If the file begins with a named ?SECTION statement, an unnamed section is considered to be present but empty.

A ?SECTION statement is followed by zero or more lines of environment definition statements and comments. A section is terminated by the next ?SECTION statement or the end of the file. Blank lines are allowed.

The environment definition statements are:

```
CONFIG [ FILE ] filename  [ filename [ filename ] ]
STARTUP [ PARAMS ] { YES | NO }
```

Both statements are optional. If more than one CONFIG or STARTUP statement is present in the section, all but the first are ignored.

```
CONFIG [ FILE ] filename  [ filename [ filename ] ]
```

defines a search list of up to three configuration files that the DSNM processes search in the order listed for configuration parameters. The default for CONFIG *filename* is *objsubvol*.DSNMCONF.

objsubvol is the subvolume on which the process program file resides.

A CONFIG statement with a blank search list is not valid. You must omit the CONFIG statement altogether to default the search list to *objsubvol*.DSNMCONF.

If a CONFIG statement is present but one or more of the file names is not fully qualified, *objsubvol* qualifies the first file name (at the local node). The remaining file names are qualified by the node, volume, and subvolume of the first file name: this is true whether the first file name is fully qualified or is partially qualified by the default *objsubvol*.

A CONFIG statement search list may specify nonexistent files; this is not an error. A nonexistent file is treated as if it were present but empty. It is ignored, and the search continues with the next configuration file. This is also true of the default DSNM configuration file (*objsubvol*.DSNMCONF), if there is no CONFIG statement for an environment.

The Tandem configuration file $SYSTEM.SYSTEM.ZDSNCONF is searched in addition to the files listed in a CONFIG statement; it is always the last file on the search list. $SYSTEM.SYSTEM.ZDSNCONF must be present.

```
STARTUP [ PARAMS ] { YES | NO }
```

determines whether DSNM configuration parameters from the startup message are used:

- If STARTUP is YES, DSNM configuration parameters in the process startup message override or supplement the value of the corresponding parameter in the DSNM configuration search list. YES is the default.
- If STARTUP is NO, DSNM configuration parameters in the process startup message are ignored.

Format of the DSNMCONF File

DSNMCONF files contain startup parameters for the various DSNM processes, subsystem and subsystem object type configuration information, and subsystem CI configuration information. A DSNMCONF file is a key-sequenced file with a ZDSN^DDL^DSNMCONF^DEF record definition. Each record represents a single instance of a parameter and contains the following fields:

Key Field	Description
SYSTEM	Identifies the Tandem node to which the parameter applies.
SUBSYS	Identifies the product to which the parameter applies; for DSNM components supplied by Tandem, the SUBSYS key field must be "DSNM."
CLASS	Identifies the class of DSNM entities to which the parameter applies.
COMPONENT	Identifies the member of the class to which the parameter applies; the component name often identifies the subsystem that the DSNM entity supports.
PARAMETER	Identifies the parameter defined by the record.
SEQUENCE	Distinguishes multiple instances of a parameter. For multivalued parameters, the valid range is 1 to 9999; for single-valued parameters, this field is blank.
VALUE	Contains the value of the parameter. Valid values depend on the particular parameter.

Searching schemes provide various ways for library procedures to retrieve values from these files. See Section 5, “DSNM Process Startup Functions,” for more information on searching schemes.

DSNMCONF Records Relevant to I Processes

The classes of DSNM configuration parameters of concern for I process development are SUBSYSTEM, SUBSYSTEM-INTERFACE-CONFIG, and CI-CONFIG. These are described in detail in the following subsections.

In addition, to integrate your subsystem and I process into a production system, your system administrator must add a COMMAND-SERVER class record and SUBSYSTEM-INTERFACE class records to the DSNMCONF file. Refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide* for definitions of these class records.

△ **Caution.** Do not alter the $SYSTEM.SYSTEM.ZDSNCONF file; however, you may override certain parameters in this file. To do so, refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide*.

SUBSYSTEM Class Records

Parameter records that specify subsystem configuration parameters have SUBSYSTEM in the CLASS field and the subsystem name in the COMPONENT field. Object type configuration is part of subsystem configuration.

Parameter records with the CLASS key field set to SUBSYSTEM specify subsystem characteristics. They are OBJTYPE, RANK, DEFAULT-OBJTYPE, DEVICETYPE, FLAGS, MANAGER, *subsystem*-MANAGER, and SUBSYSTEM-INTERFACE. Each of these is described next.

OBJTYPE

OBJTYPE describes the objects in the subsystem.

Class	Component	Parameter	Value Formats
SUBSYSTEM	*subsystem*	OBJTYPE	*objtype* [*parent-objtype* [*relative-rank*]]

objtype

is the object type name:

parent-objtype

is the object type name of the object’s parent within the subsystem object hierarchy.

relative-rank

is the object’s rank relative to other object types subordinate to the same parent. Rank determines the starting and stopping sequence of objects within a

subsystem. Objects are started in increasing rank order and stopped in decreasing order.

The default value is 0, indicating that the rank of the object is 1 greater than its parent's rank (this means that the object is one level below its parent in the subsystem object hierarchy). Nonzero relative ranks may be used to specify starting (increasing) and stopping (decreasing) orders for subordinates of the same parent type.

Default: None

Considerations: An OBJTYPE record must be entered for each object type in the subsystem.

RANK

This parameter ranks the subsystem within the DSNM hierarchy. It determines the order in which subsystem objects are brought up and down by DSNM commands.

Class	Component	Parameter	Value Formats
SUBSYSTEM	*subsystem*	RANK	*number*

Default: The default is 16.

Considerations: The valid range is 0 through 31. Rank 0 subsystem objects are started first and rank 31 last; stopping occurs in the reverse order.

Within a subsystem, starting and stopping order depends on relative rank. See the OBJTYPE parameter, described earlier.

DEFAULT-OBJTYPE

DEFAULT-OBJTYPE specifies the default object type used if an object in the subsystem cannot be resolved.

Class	Component	Parameter	Value Formats
SUBSYSTEM	*subsystem*	DEFAULT-OBJTYPE	*default-objtype* [*subordinate-objtype*]

default-objtype

is the object type.

subordinate-objtype

is the default object type for an object name in the NonStop Kernel subdevice format.

Default: None

DEVICETYPE

DEVICETYPE is the NonStop Kernel device type and subtypes, if applicable, for objects in the subsystem.

Class	Component	Parameter	Value Formats
SUBSYSTEM	*subsystem*	DEVICETYPE	*type* [*subtype* ... [*subtype*]]

Default: None.

Considerations: Up to eight subtypes are permitted.

FLAGS

FLAGS indicates how object names for the subsystem are specified and resolved.

Class	Component	Parameter	Value Formats
SUBSYSTEM	*subsystem*	FLAGS	*flagname* ... [*flagname*]

flagname

is one or more of the following values, each separated by one space:

```
[NOT] MANAGER-ALLOWED
[NOT] MGR-ALLOWED
```

indicates whether a manager is allowed with objects in the subsystem.

```
[NOT] MANAGER-REQ[UIRED]
[NOT] MGR-REQ[UIRED]
```

indicates whether a manager is required for objects in the subsystem.

```
[NOT] STAR-OBJ[ECT]-ALLOWED
[NOT] *-OBJ[ECT]-ALLOWED
```

indicates whether an asterisk (*) is accepted as a wild card for an object name in the subsystem.

```
[NOT] STAR-MANAGER-REQ[UIRED]
[NOT] *-MANAGER-REQ[UIRED]
[NOT] STAR-MGR-REQ[UIRED]
[NOT] *-MGR-REQ[UIRED]
```

indicates whether a manager process is required if a wild card (*) object is specified, thus restricting the wild card to a particular manager process.

```
[NOT] RESOLVE-OBJTYPE-WITHOUT-DNS
```

indicates whether the object name form is unique to the particular object type, and it is not necessary for the command server to use DNS for object name resolution.

`[NOT] RESOLVE-SUBOBJ-WITHOUT-DNS`

indicates whether the subordinate object is unique within a subsystem, and it is not necessary for DNS object name resolution.

`[NOT] DUMMY-DNS-MGR`
`[NOT] DUMMY-DNS-MANAGER`
`[NOT] MGR-IN-DNS-IS-DUMMY`
`[NOT] MANAGER-IN-DNS-IS-DUMMY`

indicates whether a manager name in DNS for the subsystem is used only to determine the Tandem node on which the object is located. The DNS manager name is ignored.

Default: All flags default to the NOT condition.

MANAGER

MANAGER is the unqualified file name of the subsystem manager program file, if the subsystem uses a manager of which multiple instances can be run.

This parameter is used by the command server to assist name resolution. The command server attempts to determine the subsystem of an object by comparing the file name of the manager process to this parameter value when the manager is given in a command but the subsystem is not.

The MANAGER parameter is not used to control or access the manager process.

Note. If you modify the MANAGER parameter, you must also modify the CI-CONFIG class, OBJECT-FILE parameter accordingly, if it exists.

Class	Component	Parameter	Value Formats
SUBSYSTEM	*subsystem*	MANAGER	*unqualified-filename*

Default: None.

Considerations: Only the file name is needed, not its node, volume, or subvolume. For example, specify “PATHMON” for “$SYSTEM.SYSTEM.PATHMON.”

When this form is used for automation-I-supported subsystems, the *-MGR-REQ and MGR-REQUIRED flags must be set (FLAGS parameter), and the UNDER $*manager* qualifier is required in the DSNM command syntax. You must also include CI-CONFIG class records to define the control interface process.

subsystem-MANAGER

This is the name of the subsystem manager process, if the subsystem uses a single fixed manager process for each Tandem node.

Class	Component	Parameter	Value Formats
SUBSYSTEM	*subsystem*	*subsystem*-MANAGER	*unqualified-filename*

Default: None.

Considerations: When this form is used for automation-I-supported subsystems, do not set the *-MGR-REQ and MGR-REQUIRED flags (FLAGS parameter), and do not use the UNDER $*manager* qualifier in the DSNM command syntax. If SCP is the control interface for the subsystem, you do not need to include CI-CONFIG class records.

SUBSYSTEM-INTERFACE

This parameter is the component name of the CI-CONFIG class that describes the control interface for the subsystem

Class	Component	Parameter	Value Formats
SUBSYSTEM	*subsystem*	SUBSYSTEM-INTERFACE	*CI-CONFIG-component-name*

Default: None.

Considerations: The SUBSYSTEM-INTERFACE name appears in the COMPONENT key field of the SUBSYSTEM-INTERFACE-CONFIG parameter record. It may or may not be the component of the corresponding subsystem interface process class.

process-class-CONFIG Records

Parameter records with the CLASS key field set to *process-class*-CONFIG allow one class of DSNM process to access processes in another class. Process class configuration parameters are specific to a particular type of process but not to a particular installation. You do not normally specify these parameters yourself unless you are configuring additional subsystems for DSNM support.

You must provide necessary I process configuration information so that these records can be added to the DSNMCONF file. The following process class configurations are delivered in the DSNMCONF file:

- COMMAND-SERVER-CONFIG

  Specifies fixed command server process configuration parameters (as opposed to site-specific command server configuration information contained in COMMAND-SERVER class records).

- SUBSYSTEM-INTERFACE-CONFIG

  Specifies fixed subsystem interface process configuration parameters (as opposed to site-specific configuration information contained in SUBSYSTEM-INTERFACE class records).

- CI-CONFIG

  Specifies control interface (subsystem management process or public interface management process) configuration parameters. The process class name is a name you assign to the CI and pass to _ADD^CI in your I process _STARTUP procedure.

The components associated with *process-class*-CONFIG class parameters are:

Class	Component
COMMAND-SERVER-CONFIG	*blank*
SUBSYSTEM-INTERFACE-CONFIG	CDFI PWI SCPI
CI-CONFIG	CDF-MANAGER PATHMON SCP SPOOLER-SUPERVISOR

Note: If additional subsystems are configured at your site for DSNM support, there will be associated components defined for the SUBSYSTEM-INTERFACE-CONFIG and CI-CONFIG classes.

The *process-class*-CONFIG parameters are defined next. They are PUBLIC-NAME, DEFAULT-PROCESSNAME, OBJECT-FILE, PROCESS-TYPE, MAX-PROCESSES, and OPEN-PARAMS.

PUBLIC-NAME

PUBLIC-NAME is the logical identifier for the process class to be reported in error messages about that process class.

Class	Component	Parameter	Value Formats
process-class-CONFIG	*component-name*	PUBLIC-NAME	*name*

Default: The public name defaults to the name in the COMPONENT field.

DEFAULT-PROCESSNAME

This is the default process name used to open a member of the process class if the opening process has no overriding name.

Class	Component	Parameter	Value Formats
process-class-CONFIG	*component-name*	DEFAULT-PROCESSNAME	$*process-name*

Default: None.

OBJECT-FILE

OBJECT-FILE is the program file of a process in the process class.

Class	Component	Parameter	Value Formats
process-class-CONFIG	*component-name*	OBJECT-FILE	[$*vol*.][*subvol*.]*file*

Default: None. Individual users of the class may provide their own defaults. For Tandem processes accessing subsystem managers, the default volume and subvolume is $SYSTEM.SYSTEM and $SYSTEM.SYS*nn* if the file cannot be found in $SYSTEM.SYSTEM.

PROCESS-TYPE

This is the default process name used to open a member of the process class if the opening process has no overriding name.

Class	Component	Parameter	Value Formats
process-class-CONFIG	*component-name*	PROCESS-TYPE	{ REQUESTER | SERVER}

Default: SERVER.

MAX-PROCESSES

MAX-PROCESSES is the maximum number of times a member of this process class can be opened.

Class	Component	Parameter	Value Formats
process-class-CONFIG	*component-name*	MAX-PROCESSES	*number*

Default: -1.

Considerations: A value less than zero (< 0) means there is no limit on the number of opens; a value of zero (0) means all opens to the process classes are closed after command completion; a value greater than zero (> 0) means up to the number specified process class will be kept open.

OPEN-PARAMS

OPEN-PARAMS specifies values to be used when the process class is opened.

Class	Component	Parameter	Value Format
process-class-CONFIG	*component-name*	OPEN-PARAMS	[QUALIFIER *default-qualifier*] [, NOWAIT[-DEPTH] *number*] [, OPEN-TIMEOUT *number*]

`QUALIFIER` *`default-qualifier`*

is the qualifier added to the process name when the process is opened for communication. The default is #ZSPI.

`NOWAIT[-DEPTH]` *`number`*

is the maximum concurrent operations allowed on the same open. The default value is 1.

`OPEN-TIMEOUT` *`number`*

is the time, in .01-second units, that the opener should wait for a response to an OPEN request. -1 means wait indefinitely. The default value is 0.

Default: If not configured, an appropriate internal default is supplied by each DSNM component.

Adding Subsystem Objects to the DNS Database

You must provide your system administrator with the appropriate information for adding your subsystem's objects to the DNS database. DNSCOM, the interactive interface to the DNS, and AUTOLOAD, the utility for constructing DNSCOM command (OBEY) files, are described in the *Distributed Systems Management Solutions (DSMS) System Management Guide*. The *Distributed Name Service (DNS) Management Operations Manual* provides additional instructions on using DNSCOM.

Defining an I Process as a Pathway Server

If your I process is started by PATHMON in a production environment, the process must also be defined as a Pathway server. The DSNM PATHMON configuration parameters

are in a text file named ZCPWDSMS (if your installation includes NetStatus) or ZCPWDSNM (if it does not). The following are settings for a Pathway server:

`RESET SERVER`	Resets values for all server class attributes to PATHMON defaults.
`SET SERVER AUTORESTART`	{0..32767}; *`number`* of times PATHMON attempts to restart server process after an abnormal termination. Default is 0.
`SET SERVER PRI`	{1..199}; *`priority`* at which server processes of this server class run. Default is 10 less than the priority of PATHMON.
`SET SERVER CPUS`	*`primary-cpu`*:*`backup-cpu`*
`SET SERVER LINKDEPTH`	*`maximum-number`* of concurrent links a server process can have before PATHOM directs the TCP link requests to another server process within the server class. Default is 1.
`SET SERVER MAXLINKS`	*`maximum-number`* of processes allowed in this server class. Default is unlimited.
`SET SERVER NUMSTATIC`	*`maximum-number`* of static server processes in this class. Default is 0.
`SET SERVER MAXSERVERS`	*`maximum-number`* of server processes in this server class that can run at the same time. Default is 1.
`SET SERVER PROGRAM`	$*`volume`*.*`subvol`*.*`object-file`* for the server class; this is a required parameter.
`SET SERVER STARTUP`	"*`string`*" specifying process startup parameters. For example, SET SERVER STARTUP "DSNM idev" specifies the $SYSTEM.SYSTEM.DSNM section from which the configuration file for this process is obtained.
`SET SERVER PROCESS`	$*`process-name`* within the server class that PATHMON assigns to the server process when it creates it.
`ADD SERVER`	*`server-class-name`*; names server class and adds its definitions to PATHMON control file.
.	
.	
`START SERVER`	*server-class-name*; starts NUMSTATIC number of server processes for this server class.

Defining server class configurations is described in detail in the *Pathway System Management Reference Manual*, if you are running NonStop Kernel release D30.01, or in the *NonStop TS/MP and Pathway System Management Guide*, if you are running NonStop Kernel release D30.02 or later.

Scope of This Section

This section describes DSNMCom, the I process test utility. It provides the syntax descriptions for the DSNMCom commands and parameters.

What is DSNMCom?

DSNMCom is a user interface to DSNM. It enables you to bypass NetCommand and send DSNM commands directly to a started process (such as a command server or an I process). This lets you test your I process without having to set up a DSNM environment. DSNMCom must be licensed before you can use it.

DSNMCom supports the DSNM commands described in Section 2, "DSNM Commands." If you are using DSNMCom to send commands directly to an I process, object names must be fully qualified, because no name resolution takes place.

Before You Run DSNMCom

Before you run DSNMCom:

- Configuration records for the subsystem and its subsystem interface process must be added to a DSNMCONF file (see Section 6, "Configuring a New Subsystem Into DSNM").
- The process(es) you wish to communicate with must be started before DSNMCom can open them. For example, to start the I process associated with the SPIFFY subsystem referred to in the examples that follow, type the following RUN command:

```
> RUN $DSNM.IDEV.SPIFI/NAME $SPFI,NOWAIT/TESTMODE 1, &
     CONFIG $DSNM.IDEV.DSNMCONF
```

  The TESTMODE and CONFIG process parameters are discussed in Section 5, "DSNM Process Startup Functions."

DSNMCom Command Syntax

DSNMCom can accept input interactively or from a specified file. To use the following command syntax, the object code for DSNMCom must reside in one of the subvolumes contained in your #PMSEARCHLIST file (if it does not, add the code to your TACLCSTM file):

```
DSNMCOM [ / run-option [ , run-option ] .../ ]
        [ DSNM [ section-name ] | CONFIG [ filename ] ]
        [[,] $process-name ]
        [[;] [ . ] command ]
```

run-option

is any run option for the TACL RUN command. Run options must be separated by commas and set off in the command line by slashes (/). See the *TACL Reference Manual* for descriptions of valid run options.

Two run options most often used with DSNMCom are:

```
IN filename
OUT filename
```

IN *filename*

causes DSNMCom to read and execute commands located in the specified file.

If you omit IN *filename*, DSNMCom uses the input file in effect for the current TACL process: usually, the home terminal.

OUT *filename*

causes DSNMCom to send its output to the specified file.

If you omit OUT *filename*, the output is directed to the output file in effect for the current TACL process: usually, the home terminal.

If *filename* does not exist, an EDIT file is created.

DSNM *section-name*

initializes the DSNMCom subsystem and object tables using the configuration file pointed to in the named *section-name* of $SYSTEM.SYSTEM.DSNM. If you do not specify a section name (blank value), the blank section of $SYSTEM.SYSTEM.DSNM is used.

CONFIG *filename*

initializes the DSNMCom subsystem and object tables using the named configuration file. Only one configuration file can be specified. If do not specify a configuration file (blank value), the blank section of $SYSTEM.SYSTEM.DSNM is used.

▲ **WARNING.** DSNMCom and the process(es) you are testing must all be using the same DSNMCONF file for the subsystem, CI, and I process configuration parameters. If you are testing an I process, you must specify the same DSNMCONF file in both your I process RUN command and your DSNMCom RUN command; otherwise, results are unpredictable.

If you specify more than one configuration file in your I process RUN command (for instance, if you are maintaining your subsystem-specific configuration parameters separately from your installation's production DSNM configuration parameters), you must provide DSNMCom with the configuration file that contains the subsystem-specific configuration information.

$*process-name*

is the name of the process to which DSNM sends commands.

If you omit $*process-name*, you must open the process with the DSNMCom OPEN command before issuing any DSNM commands (see “The DSNMCom Commands” on page 7-5).

$*process-name* must be preceded by a comma if you specify either DSNM or CONFIG with a blank value.

command

is either one of the DSNMCom commands listed in Table 7-1, or a DSNM command. Separate the command from the $*process-name* by a semicolon or at least one space.

The DSNMCom Prompt

The DSNMCom prompt is one of the following:

- When you enter DSNMCOM at your TACL prompt without specifying a process name to be opened, the following prompt appears:

  ```
  DSNMCom >
  ```

- Once you open a process, either by including the process name in your run command or by issuing an OPEN command from DSNMCom, the following prompt appears:

  ```
  DSNM $process-name >
  ```

- If DSNMCom is unable to open the specified process, the following prompt appears:

  ```
  DSNM $process-name (filesystem-error) >
  ```

Running DSNMCom Interactively

You can run DSNMCom interactively in one of the following ways:

- By executing a DSNM command through DSNMCom at the TACL prompt. For example:

  ```
  > DSNMCOM CONFIG $DSNM.IDEV.DSNMCONF $SPFI &
       STOP VALVE PRT3 UNDER $SMGR
  >
  ```

- By entering DSNMCom and executing DSNM commands at the DSNMCom prompt. For example:

  ```
  > DSNMCOM CONFIG $DSNM.IDEV.DSNMCONF $SPFI
  DSNM $spfi > STOP VALVE PRT3 UNDER $SMGR
  DSNM $spfi >
  ```

Running DSNMCom From an Input File

Use the IN file name option with the RUN command to provide DSNMCom with a set of commands to execute. For example, to execute a set of DSNM commands in file EXCMDS, using the configuration file pointed to in section TEST of $SYSTEM.SYSTEM.DSNM, type:

```
> DSNMCOM /IN EXCMDS/ DSNM TEST $process-name
      .
    DSNM command output
      .
>
```

The end of file (EOF) of the input file terminates the DSNMCom process; control of the terminal returns to TACL. (EOF is the same as typing EXIT.)

To execute the commands located in file EXCMDS and send the output to printer $HT1, type:

```
> DSNMCOM /IN EXCMDS,OUT $S.#HT1,NOWAIT/ DSNM TEST &
    $process-name
>
```

The Comment Character, COMMENT-CHAR

To have input recognized as comments by DSNMCom, you must precede each comment line with one COMMAND-CHAR character and one COMMENT-CHAR character. By default, both are the period (.) and both are set using the DSNMCom SET command, discussed later in this section.

To have an input line be interpreted as a comment by DSNMCom, use the following syntax:

```
COMMAND-CHAR COMMENT-CHAR comment-text
```

With both COMMAND-CHAR and COMMENT-CHAR set to a period (.), an example would be the following:

```
..setting PAID is optional
```

Using the Break Key

Use the Break key to stop a command that is listing information to the screen. DSNMCom then returns to the DSNMCom prompt.

If you press the Break key from within DSNMCom when a listing command is not executing, control of the terminal returns to TACL. DSNMCom continues to run in the background. From the TACL prompt, you can:

- Type PAUSE to return to DSNMCom.
- Type STOP to stop the running DSNMCom process.

Setting Security Parameters in DSNMCom

To set security parameters in DSNMCom, use the DSNMCom SET command. To display the current settings before setting or changing them, use the DSNMCom SHOW command, followed by the parameter name. The SHOW command without a parameter name displays the current settings of all these parameters. The security parameters are discussed with the DSNMCom SET command definition, later in this section.

The DSNMCom Commands

With the exception of the FC command, DSNMCom commands must be preceded by the COMMAND-CHAR character; otherwise, the command is not recognized. No leading spaces are allowed.

The default COMMAND-CHAR character is the period (.). You can change the value of COMMAND-CHAR with the DSNMCom SET command, which is discussed later in this section. Table 7-1 lists the DSNMCom commands.

Table 7-1. DSNMCom Commands

Command	Function
CLOSE	Closes the current server.
EXIT	Closes all files and terminates DSNMCom.
FC	Allows you to edit and reexecute the previous command line.
HELP	Displays all DSNMCom commands or the syntax of a particular command.
OPEN	Opens a process.
QUIT	Closes all files and terminates DSNMCom (same as EXIT).
RESET	Returns all settable parameters to their default values.
SET	Sets DSNMCom parameters.
SHOW	Displays the current settings of DSNMCom parameters.

CLOSE Command

The CLOSE command closes the current server.

```
CLOSE
```

EXIT Command

The EXIT command terminates DSNMCom and returns control to the command interpreter.

```
EXIT
```

Considerations

- The DSNMCom process terminates when it reads the end of file (EOF) of an input file: you do not have to end an input command file with an EXIT command.
- Entering Ctrl/Y at the terminal is the same as EOF. If you type Ctrl/Y at the DSNMCom prompt, DSNMCom terminates.
- The DSNMCom QUIT command is synonymous with the EXIT command.

FC Command

FC (fix command) allows you to modify and resubmit the last command line entered. The FC command is not recognized if preceded by a period (.).

```
FC
```

Considerations

The FC subcommands (R, I, and D) are the same as those used for the TACL FC command. See the *TACL Reference Manual* for further description.

HELP Command

The HELP command displays DSNMCom command and parameter names or the syntax of a particular DSNMCom command. It also provides limited usage information for commands and parameters.

```
HELP [ / OUT filename / ] [ command or paramname ]
```

OUT *filename*

is the output device for the listing. If omitted, the output is sent to your home terminal.

command

is the name of the DSNMCom command whose syntax you want to see.

paramname

is the name of the DSNMCom parameter about which you want to see information. The DSNMCom parameters are listed in the “SET Command” subsection, later in this section.

OPEN Command

The OPEN command opens the specified process.

```
OPEN $process-name
```

$*process-name*

is the name of the process to which you want to send DSNM commands.

Consideration

A previously opened process (if any) is closed upon successful completion of the OPEN command.

QUIT Command

The QUIT command closes all files and terminates DSNMCom; it is synonymous with the EXIT command.

```
QUIT
```

Consideration

See the EXIT command considerations.

RESET Command

The RESET command returns all settable parameters to their default values. You might find this command useful to quickly reset all the DSNMCom parameters to their defaults before resetting just one parameter (for a specific test). For a specific test, use the RESET command to restore all the default settings, then use the SET command to set a single parameter.

```
RESET
```

SET Command

The SET command allows you to set DSNMCom parameters, most of which are security attributes. The LICENSED, LOGGED-ON, REMOTE, SEND-SECINFO, and TYPED-OUTPUT parameters can be set to YES, NO, TRUE, FALSE, ON, or OFF.

These security parameters allow you to establish a simulated test environment where processes and commands appear to have certain characteristics or security attributes. This allows you to test your I process or server process without establishing an entire DSNM environment.

```
SET paramname paramvalue
```

The values of *paramname* and *paramvalue* appear in Table 7-2, which lists the DSNMCom SET parameters.

Table 7-2. DSNMCom SET Parameters

Parameter	Function	Default
COMMAND-CHAR	Required first character of each line that contains a DSNMCom command or a comment (except for the FC command).	. (period)
COMMENT-CHAR	Character that must follow COMMAND-CHAR for a comment line.	. (period)
LICENSED	When set to TRUE, makes DSNM commands appear to be from a licensed requester.	FALSE
LOGGED-ON	When set to TRUE, makes DSNM commands appear to be sent from a logged-on requester.	FALSE
PAID	Process accessor ID value of an open request sent to DSNM.	None
REMOTE	When set to TRUE, makes DSNM commands appear to be sent from a remote process or a process created by a remote process.	FALSE
SEND-SECINFO	When set to TRUE, DSNMCom sends commands with security data consistent with that sent by the DSNM CMDSVR process.	FALSE
TYPED-OUTPUT	When set to TRUE, DSNMCom displays VTY type information.	FALSE
USER	NonStop Kernel user ID of the user under which the command is to be processed.	None

COMMAND-CHAR

COMMAND-CHAR is the character that must appear as the first character of each line that contains a DSNMCom command or a comment. When setting COMMAND-CHAR, enter the new character without quotes. For example, to set the command character to the slash (/), use the following:

```
.SET COMMAND-CHAR /
```

Before processing the command in this example, COMMAND-CHAR was set to a period (.); notice a period at the beginning of the command line. After DSNMCom processes this command, COMMAND-CHAR becomes the slash. All subsequent DSNMCom commands and comments must now begin with a slash (until the next setting of this character or a DSNMCom RESET command).

COMMENT-CHAR

COMMENT-CHAR is the character that must follow COMMAND-CHAR for the remainder of the line to be ignored (to be interpreted as a comment). When setting COMMENT-CHAR, enter the new character without quotes. For example, to set the comment character to the slash (/) use the following:

```
.SET COMMENT-CHAR /
```

After DSNMCom processes this command, COMMENT-CHAR becomes the slash (until the next setting of this character or a DSNMCom RESET command).

LICENSED

When the LICENSED parameter is set to TRUE, all DSNM commands sent to the server appear to be from a licensed requester. The LICENSED parameter is only effective in this manner when the DSNMCom process is licensed and the security parameter SEND-SECINFO is also TRUE.

LOGGED-ON

When LOGGED-ON is set to TRUE, all DSNM commands sent to the server appear to be from a logged-on requester. The LOGGED-ON parameter is only effective in this manner when the DSNMCom process is licensed and the security parameter SEND-SECINFO is also TRUE.

PAID (Process Accessor ID)

The process-accessor-id (PAID) is the user ID under which the process gains access to system objects: for example, files. The value of PAID is sent as the process-accessor-id of the open operation over which the request was sent to DSNM.

The PAID parameter is only effective in this manner when the DSNMCom process is licensed and the security parameter SEND-SECINFO is set to TRUE.

PAID can be set to *groupname.username* or *groupnumber,usernumber.*

REMOTE

When REMOTE is set to TRUE, the command sent to a server appears to be from a remote process or one that was created by a remote process and has not logged on locally. For a TRUE setting of REMOTE to have any effect, the DSNMCom process must be licensed, and the security parameter SEND-SECINFO is also set to TRUE.

SEND-SECINFO

When SEND-SECINFO is set to TRUE, DSNMCom sends commands with security information (internal DSNMCom security information: ZDSN^TKN^SECURITY^INFO) consistent with that sent by the DSNM CMDSVR process. DSNMCom must be licensed for SEND-SECINFO (when set to TRUE) to support valid security testing.

When SEND-SECINFO is FALSE, DSNMCom sends logon information (ZDSN^TKN^LOGON^INFO) in the same manner as an external DSNM requester (such as NonStop NET/MASTER). Logon information is only sent if parameter USER is set.

If you are testing an I process or a secondary command server, use TRUE for SEND-SECINFO. If you are testing a primary command server, use FALSE for SEND-SECINFO.

TYPED-OUTPUT

When TYPED-OUTPUT is set to TRUE, DSNMCom displays the VTY type information for each line of output from a server response.

USER

USER is the NonStop Kernel user ID of the user under which the command is to be processed by the server. USER can be set to *groupname.username* or *groupnumber,usernumber.*

Consideration

USER is allowed only if DSNMCom is licensed.

SHOW Command

The SHOW command displays the current values of all the DSNMCom parameters. If you follow the SHOW command with a parameter name, DSNMCom displays the current setting of that parameter.

```
SHOW [ paramname ]
```

Considerations

None.

Executing DSNM Commands

Any command you enter from within DSNMCom other than CLOSE, EXIT, FC, HELP, OPEN, QUIT, RESET, SET, or SHOW is assumed to be a DSNM command.

You can send DSNM commands directly from the DSNMCom RUN command line. For example:

```
> DSNMCOM CONFIG $DSNM.IDEV.DSNMCONF $SPFI STATUS REACTOR * &
     UNDER $SMGR, DOWN
SPIFFY   CHAMBER COMPOUND3 UNDER $SMGR Down
SPIFFY   BOILER STUFFX UNDER $SMGR Down
SPIFFY   VALVE  FORMULAY UNDER $SMGR Down
SPIFFY   BOILER INGREDTA UNDER $SMGR Down
SPIFFY   BOILER INGREDTC UNDER $SMGR Down
SPIFFY   VALVE  XXX UNDER $SMGR Down
>
```

Or you can send DSNM commands from within DSNMCom. For example

```
> DSNMCOM CONFIG $DSNM.IDEV.DSNMCONF $SPFI
DSNMCom - T9216D30 12FEB95
Copyright Tandem Computers Incorporated 1995
DSNM $spfi > STATUS REACTOR PURPLE UNDER $SMGR, NOT-UP
SPIFFY   BOILER ELEMENT1 UNDER $SMGR Pending
SPIFFY   BOILER ELEMENT2 UNDER $SMGR Pending
SPIFFY   BOILER ELEMENT3 UNDER $SMGR Pending
SPIFFY   VALVE  MIX3 UNDER $SMGR Pending
SPIFFY   CHAMBER COMPOUND2 UNDER $SMGR Pending
SPIFFY   CHAMBER COMPOUND3 UNDER $SMGR Down

DSNM $spfi > STATUS REACTOR AMBER UNDER $SMGR, NOT-DOWN
SPIFFY   BOILER INGREDTB UNDER $SMGR Up
SPIFFY   VALVE  YYY UNDER $SMGR Up
SPIFFY   VALVE  ZZZ UNDER $SMGR Pending
SPIFFY   CHAMBER AAA UNDER $SMGR Pending
SPIFFY   CHAMBER BBB UNDER $SMGR Up
SPIFFY   CHAMBER CCC UNDER $SMGR Pending

DSNM $spfi> EXIT
>
```

DSNMCom Messages

DSNMCom produces the following messages, which are displayed on the OUT listing device.

```
ERROR -- no server open:  use OPEN <process>
```

Cause. You tried to issue a DSNM command without first opening a process.

Effect. You are returned to the DSNMCom prompt.

Recovery. Use the OPEN command to open a started process; then resubmit the command.

```
ERROR -- Invalid process name
```

Cause. You provided an invalid process name to DSNMCom.

Effect. You are returned to the DSNMCom prompt.

Recovery. Use the OPEN command and provide a valid process name.

```
ERROR -- process name expected
```

Cause. You tried to open a process without providing a process name.

Effect. You are returned to the DSNMCom prompt.

Recovery. Reenter the command with a valid process name.

```
File System error nn ON filename
```

Cause. A file system error occurred when DSNMCom attempted a WRITEREAD to either the input file, the output file, or the currently open process file.

Effect. If the error occurred on the input file, the message is written to the output file and DSNMCom abnormally ends.

If the error occurred on the output file, the message is written to the home terminal and DSNMCom abnormally ends.

If the error occurred on the currently open process file, the message is written to the output file and the process file is closed. Any subsequent SEND commands result in an “ERROR -- no server open” message until an OPEN command is successfully processed.

Recovery. See the *Guardian Procedure Errors and Messages Manual* for a description of the specific file system error and the appropriate recovery action.

```
OPEN ERROR nn ON filename
```

Cause. DSNMCom was unable to open the specified file.

Effect. If the error occurred on the input or output file, DSNMCom abnormally ends after writing a brief description of the error to the home terminal.

If the error occurred on the process file, the process file remains closed; any previously opened process file remains open.

Recovery. See the *Guardian Procedure Errors and Messages Manual* for a description of the specific file system or sequential I/O (SIO) error and the appropriate recovery action.

```
ERROR -- Section name too long
```

Cause. DSNMCom was unable to open the specified STARTUP parameter value in the $SYSTEM.SYSTEM.DSNM file (a section named env-name) during the DSNM initialization process because the specified section name is too long. The maximum acceptable length for env-name (ZDSN-MAX-PARAMNAME) is 32 bytes.

Effect. A fatal error is reported, and DNSM is not started.

Recovery. Correct and reissue the command. Refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide* for information on DSNM process startup message parameters.

```
ERROR -- Expecting <space> <comma> or <semi-colon>
```

Cause. The command did not include an expected character.

Effect. DSNMCom is not started up.

Recovery. Correct and reissue the command.

```
ERROR -- DSNMCom can't continue
```

Cause. During the initialization of DSNM, inappropriate configuration input was submitted, or no configuration was submitted.

Effect. DSNM and DSNMCom cannot be started.

Recovery. Check your configuration files and try the DSNM initialization again. If the problem persists, contact your Tandem representative. Be prepared to describe the problem in detail as recommended in “What to Prepare Before Contacting Your Tandem Support Representative” on page B-1.

```
ERROR -- unrecognized text following command
```

Cause. You provided text that is not recognized or supported after the DSNMCom command.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
ERROR -- expecting one of: ON, OFF, TRUE, FALSE, YES or NO
```

Cause. You provided a parameter value that is not recognized or supported.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
ERROR -- invalid value for parameter
```

Cause. You provided a parameter value that is not recognized or supported; DSNMCom was expecting a single character.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
ERROR -- invalid username
```

Cause. You provided a user name that is unrecognizable, not supported, or in an improper format.

Effect. The command is not executed.

Recovery. Correct and reissue the command. The user name must be in the format *groupnumber.usernumber*, where both *groupnumber* and *usernumber* are no more than eight characters and begin with a letter (A to Z or a to z). Also, a period (.) must appear between *groupnumber* and *usernumber*.

```
ERROR -- invalid userid
```

Cause. You provided a user ID that is unrecognizable, not supported, or in an improper format.

Effect. The command is not executed.

Recovery. Correct and reissue the command with a valid user ID.

```
ERROR -- Unable to obtain userid
```

Cause. Userid is out of bounds or there is an I/O error on $SYSTEM.SYSTEM.USERID.

Effect. The command is not executed.

Recovery. Check USERID or contact your system manager.

```
ERROR -- Unable to obtain username
```

Cause. Username is out of bounds or there is an I/O error on $SYSTEM.SYSTEM.USERID.

Effect. The command is not executed.

Recovery. Check USERNAME or contact your system manager.

```
ERROR -- non-existent user
```

Cause. You specified a user ID / user name that is undefined.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
ERROR -- expecting a valid userid or username
```

Cause. You provided a user ID or user name in a DSNMCom SET command that is not recognized or supported.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
ERROR -- expecting a valid parameter name
```

Cause. You misspelled a parameter or issued a SET or SHOW command with a parameter name that is not supported.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
ERROR -- invalid DSNMCom command
```

Cause. DSNMCom is unable to recognize the command because the command line did not begin with the current COMMAND-CHAR character, and the command is not a valid DSNM command.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
DSNM error:  error-text ON filename
```

Cause. A DSNM error occurred during the file operation.

Effect. See the description in the *error-text* portion of the message.

Recovery. See the *error-text* portion of the message.

```
SSGETTKN Error nn on LOGON^INFO token
```

Cause. An SPI error occurred.

Effect. The command is not executed

Recovery. Check the SPI error identified in the message and refer to the SPI error documentation in the *SPI Programming Manual*.

```
Warning:  SSPUTTKN Error nn adding the SECURITY-INFO token
```

Cause. An SPI error occurred.

Effect. The command is sent without the security token.

Recovery. Check the SPI error identified in the message and refer to the SPI error documentation in the *SPI Programming Manual*.

```
Warning:  Error nn trying to construct SECURITY-INFO token
SECURITY-INFO not added to command buffer
```

Cause. An SPI error occurred.

Effect. The command is sent without the security token.

Recovery. Check the SPI error identified in the message and refer to the SPI error documentation in the *SPI Programming Manual*.

```
SECURITY-INFO not added to command buffer
```

Cause. An SPI error occurred.

Effect. The command is sent without the security token.

Recovery. No recovery action required.

```
no server open
```

Cause. The CLOSE command was given when no server was open.

Effect. The command is not executed.

Recovery. No recovery action needed.

DSNM Parser Errors

The following errors may be generated by the DSNM parser, which interprets DSNM commands before they are executed.

```
Command unrecognizable
```

Cause. You misspelled a command or issued a command that is not supported.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
Exceeded max objects
```

Cause. Your command included more objects than are allowed in a single command.

Effect. The command is not executed.

Recovery. Break the command up into two or more commands.

```
Exceeded max paren levels
```

Cause. You nested parentheses beyond the maximum allowable depth.

Effect. The command is not executed.

Recovery. Simplify the command or break it into two or more commands, if necessary.

```
Exceeded param space
```

Cause. You entered too many parameters or an excessively long parameter.

Effect. The command is not executed.

Recovery. Simplify the command or break it into two or more commands, if necessary.

```
Invalid option
```

Cause. You specified an option that is not valid for any command.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
Invalid option for this command
```

Cause. You specified an option that is not valid for this command.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
Name too long
```

Cause. You entered a name that is longer than the maximum valid length.

Effect. The command is not executed.

Recovery. Correct the name and reissue the command.

```
No operands
```

Cause. You issued a command that requires operands, but did not specify any.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
Object type table error
```

Cause. A consistency error was encountered in the parser object type table.

Effect. Commands cannot be correctly interpreted.

Recovery. Contact your Tandem representative. Be prepared to describe the problem in detail as recommended in “What to Prepare Before Contacting Your Tandem Support Representative” on page B-1.

```
Param datatype table error
```

Cause. A consistency error was encountered in the parser parameter data type table.

Effect. Commands cannot be correctly interpreted.

Recovery. Contact your Tandem representative. Be prepared to describe the problem in detail as recommended in “What to Prepare Before Contacting Your Tandem Support Representative” on page B-1.

```
Reserved word misplaced
```

Cause. A keyword, subsystem name, or object type was out of place.

Effect. The command is not executed.

Recovery. Correct and reissue the command. If the error was caused because an object name is the same as a keyword, subsystem name, or object type, enclose it in quotation marks.

```
Subsys table error
```

Cause. A consistency error was encountered in the parser subsystem table.

Effect. Commands cannot be correctly interpreted.

Recovery. Contact your Tandem representative. Be prepared to describe the problem in detail as recommended in “What to Prepare Before Contacting Your Tandem Support Representative” on page B-1.

```
Syntax error
```

Cause. The command contains a serious syntax error.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
Unbalanced parens
```

Cause. Your command includes parentheses that are incorrectly paired.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

```
Unexpected end
```

Cause. The command did not include all required and expected information.

Effect. The command is not executed.

Recovery. Correct and reissue the command.

A DSNM Library Services

Scope of This Appendix

This appendix provides the following information (as applicable) for each define, literal, procedure, global variable, and structure template listed in Table A-1 below:

- Description
- Syntax
- Parameter descriptions
- Considerations (additional information)
- Examples

Table A-1. DSNM Library Services (page 1 of 4)

Identifier	Define	Literal	Procedure	Global Variable	Structure Template	Miscellaneous
_ADD^CI			X			
_ADD^SUBSYS			X			
_ALLOFF	X					
_ALLON	X					
_ALLON^TURNOFF	X					
_ANYOFF	X					
_ANYON	X					
_ANYON^TURNOFF	X					
_APPEND^OUTPUT			X			
_BITDEF	X					
_CANCEL^SEND^CI			X			
_CANCEL^TIMEOUT			X			
_CI^DEF	X					
_CI^FILENUM	X					
_CI^ID	X					
_CI^IDPOINTER	X					
_CI^LASTERROR	X					
_CI^REPLYADDRESS	X					
_CI^REPLYLENGTH	X					
_CI^REPLYTAG	X					
_CLOSE^CI			X			
_COMMAND^CONTEXT^HEADER	X					
_COMMAND^PROC						Name of initial command thread procedure

Table A-1. DSNM Library Services (page 2 of 4)

Identifier	Define	Literal	Procedure	Global Variable	Structure Template	Miscellaneous
_COMMAND^TERMINATION^PROC						Name of thread termination procedure
_COMPILED^IN^TESTMODE		X				
_DEALLOCATE^LIST			X			
_DELETE^LM			X			
_DEPOSIT	X					
_DISPATCH^THREAD	X					
_DSNMCONF^PARAMS					X	
_EMPTY^LIST	X					
_EMS^EVENT^CRITICAL		X				
_EMS^EVENT^FATAL		X				
_EMS^EVENT^INFO		X				
_END^THREAD^PROC	X					
_END^THREAD^TERMINATION^PROC	X					
_EV^CANCEL		X				
_EV^CONTINUE		X				
_EV^IODONE		X				
_EV^STARTUP		X				
_EV^TIMEOUT		X				
_EXTRACT	X					
_FIRST^LM			X			
FOBJECT						Name assigned to formatted object structure
_FOBJECT^INIT			X			
_GET^LM			X			
_GET^PARAM			X			
_GET^PROCESS^PARAM			X			
_INITIALIZE^LIST			X			
_INPUT						Name assigned to command context input area
_INPUT^DEF					X	
_INPUT^LM^HEADER	X					
_ISNULL	X					
_JOIN^LIST			X			
KDSNDEFS						Source file for definitions and declarations
_LAST^CI^ID	X					

Table A-1. DSNM Library Services (page 3 of 4)

Identifier	Define	Literal	Procedure	Global Variable	Structure Template	Miscellaneous
_LAST^EVENTS				X		
_LAST^LM			X			
_LAST^TIMEOUT^TAG				X		
_LIST	X					
_LISTPOINTER	X					
_MEMBERSOF^LIST	X					
_MOVE^LIST			X			
_NOTNULL	X					
_NULL		X				
_NULL^LIST			X			
OBJECTLIST						Name assigned to input and output object lists
_OFF	X					
_ON	X					
_OPEN^CI			X			
_OUTPUT						Name assigned to command context output area
_OUTPUT^DEF					X	
_OUTPUT^LM^HEADER	X					
_POP^LM			X			
_POP^THREAD^PROCSTATE			X			
_PREDECESSOR^LM			X			
_PRIVATE^THREAD^EVENT	X					
_PROCESS^PARAMS				X		
_PUSH^LM			X			
_PUSH^THREAD^PROCSTATE			X			
_PUT^LM			X			
_RC^ABORT	X					
_RC^NULL		X				
_RC^STOP		X				
_RC^TYPE	X					
_RC^WAIT		X				
_REAL^LAST^EVENTS	X					
_RELEASE^OUTPUT	X					
_REPORT^INTERNAL^ERROR			X			
_REPORT^STARTUP^ERROR			X			
_RESTORE^THREAD^AND^DISPATCH	X					

Table A-1. DSNM Library Services (page 4 of 4)

Identifier	Define	Literal	Procedure	Global Variable	Structure Template	Miscellaneous
_SAVE^THREAD^AND^DISPATCH	X					
_SEND^CI			X			
_SET^THREAD^PROC	X					
_SET^TIMEOUT			X			
_SIGNAL^EVENT	X					
_STARTUP						User-supplied initialization procedure
_STARTUP^MODE						User-supplied startup procedure
_ST^INITIAL		X				
_ST^MIN^THREAD^STATE		X				
_SUBSYS^DEF					X	
_SUCCESSOR^LM			X			
_THREAD^CONTEXT^ADDRESS				X		
_THREAD^PROC	X					
_THREAD^STATE	X					
_THREAD^TERMINATION^CODE	X					
_THREAD^TERMINATION^PROC	X					
_TURNOFF	X					
_TURNON	X					
_UNGET^LM			X			
_UNPOP^LM			X			
_XADR^EQ	X					
_XADR^NEQ	X					

_ADD^CI

_ADD^CI retrieves CI configuration information from the DSNM configuration file (DSNMCONF). It allocates the memory for and completes a predefined CI configuration structure with this information. It then returns the address of the completed structure.

You must declare an extended pointer to a structure defined by _CI^DEF in your global definitions for each CI with which your subsystem communicates.

You must call _ADD^CI in your _STARTUP procedure for each CI class with which your I-process communicates.

```
@ci-config := _ADD^CI ( ciname
                      ,[ error ]
                      ,[ error-filename ] );
```

ci-config output

INT .EXT ! (_CI^DEF) !

receives the address of the _CI^DEF-defined CI configuration structure that _ADD^CI completes with configuration parameter values for the specified CI. If an error occurs or no configuration is found for this CI process class, a null value is returned.

ciname input

STRING .EXT

is the process class name associated with the CI. The string must be blank or null terminated.

The process class name of the CI is the name in the COMPONENT field of the CI-CONFIG class records in the DSNMCONF file that specifies the CI configuration parameters. This name is arbitrary; by custom, the object file name of the subsystem manager is the logical name of the process class (for example, PATHMON or SCP).

error output

INT .EXT

is the ZDSN or file-system error, if an error occurs.

error-filename output

STRING .EXT ! (ZDSN^DDL^OBJNAME^DEF) !

is the name of the DSNM configuration file associated with the returned *error* value.

Example

```
< in global definitions >

INT .EXT scp (_CI^DEF);
INT .EXT snaxcdf (_SUBSYS^DEF);

STRING .scpclass[0:ZDSN^MAX^CICLASS-1] := [“SCP “];
STRING .cdf[0:ZDSN^MAX^SUBSYS-1] := [“SNAXCDF “];

< within _STARTUP procedure >

   .
   .
  IF _ISNULL (@scp := _ADD^CI (scpclass)) THEN
    RETURN ZDSN^ERR^INTERNAL^ERR;
  IF _ISNULL (@snaxcdf := _ADD^SUBSYS (cdf)) THEN
    RETURN ZDSN^ERR^INTERNAL^ERR;
   .
   .

< within _COMMAND^PROC procedure >

   .
   .
  CALL _OPEN^CI (scp, ... );
   .
   .
```

_ADD^SUBSYS

_ADD^SUBSYS retrieves subsystem and object type configuration information from the DSNM configuration file (DSNMCONF). It allocates the memory for and completes a predefined subsystem configuration structure with this information. It then returns the address of the completed structure.

You must declare an extended pointer to a subsystem configuration structure defined by _SUBSYS^DEF in your global definitions for each subsystem your I-process handles.

You must call _ADD^SUBSYS in your _STARTUP procedure for each subsystem your I-process handles.

```
@ss-config := _ADD^SUBSYS ( ssname
                          ,[ error ]
                          ,[ error-filename ] );
```

ss-config output

INT .EXT ! (_SUBSYS^DEF) !

receives the address of the _SUBSYS^DEF-defined subsystem configuration structure that _ADD^SUBSYS completed with configuration parameter values for the specified subsystem. If an error occurs or no configuration is found for this subsystem, a null value is returned.

ssname input

STRING .EXT

is the subsystem name. The subsystem name is the name in the COMPONENT field of the SUBSYSTEM class records in the DSNMCONF configuration file that specifies the subsystem configuration parameters.

error output

INT .EXT

is the ZDSN or file-system error, if an error occurs.

error-filename output

STRING .EXT ! (ZDSN^DDL^OBJNAME^DEF) !

is the name of the DSNM configuration file associated with the returned *error* value.

Example

```
< in global definitions >

INT .EXT scp (_CI^DEF);
INT .EXT snaxcdf (_SUBSYS^DEF);

STRING .scpclass[0:ZDSN^MAX^CICLASS-1] := [“SCP “];
STRING .cdf[0:ZDSN^MAX^SUBSYS-1] := [“SNAXCDF “];

< within _STARTUP procedure >

   .
   .
  IF _ISNULL (@scp := _ADD^CI (scpclass)) THEN
    RETURN ZDSN^ERR^INTERNAL^ERR;
  IF _ISNULL (@snaxcdf := _ADD^SUBSYS (cdf)) THEN
    RETURN ZDSN^ERR^INTERNAL^ERR;
   .
   .
```

_ALLOFF

_ALLOFF is a Boolean define statement that is TRUE if every one-bit of *bit-mask* is off in *int-exp*. TRUE is nonzero, not necessarily -1.

The _ALLOFF function is the same as the _OFF function. It is more descriptive to use _ALLOFF when testing more than one bit.

```
_ALLOFF ( int-exp , bit-mask )
```

int-exp input

INT:value

is the INT expression being compared with *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-exp* to test.

Example

The following example tests if both bits 9 and 11 in var are off:

```
INT var;
LITERAL evta = %20;   !evta.<11> on
LITERAL evtb = %100;  !evtb.<9> on

IF _ALLOFF ( var, evta + evtb )
  THEN ...       !both var.<9> and var.<11> are zero
  ELSE ...;      !at least one of var.<9> and var.<11> is one
```

_ALLON

_ALLON is a Boolean define statement that is TRUE if every one-bit of *bit-mask* is on in *int-exp*. TRUE is nonzero, not necessarily -1.

```
_ALLON ( int-exp , bit-mask )
```

int-exp input

INT:value

is the INT expression being compared with *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-exp* to test.

Examples

The following example tests if both bits 9 and 11 in var are on:

```
INT var;
LITERAL evta = %20;   !evta.<11> on
LITERAL evtb = %100;  !evtb.<9> on

IF _ALLON ( var, evta + evtb )
  THEN ...    !both var.<9> and var.<11> are one
  ELSE ...;   !at least one of var.<9> and var.<11> is
              !zero
```

The following example tests if the event _EV^IODONE caused the current dispatch of the thread:

```
IF _ALLON (_LAST^EVENTS, _EV^IODONE)
  THEN ...;
```

_ALLON^TURNOFF

If every one-bit of *bit-mask* is on in *int-var*, _ALLON^TURNOFF returns TRUE and turns off every one-bit of *int-var* that is on in *bit-mask*. TRUE is nonzero, not necessarily -1.

```
_ALLON^TURNOFF ( int-var , bit-mask )
```

int-var input/output

INT:ref

is the variable compared with *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-var* to turn off.

_ANYOFF

_ANYOFF is a Boolean define statement that is TRUE if any one-bit of *bit-mask* is off in *int-exp*. TRUE is nonzero, not necessarily -1.

```
_ANYOFF ( int-exp , bit-mask )
```

int-exp input

INT:value

is the INT expression compared with *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-exp* to test.

Example

The following example tests if at least one of bits 9 and 11 in var are off:

```
INT var;
LITERAL evta = %20;   !evta.<11> on
LITERAL evtb = %100;  !evtb.<9> on

IF _ANYOFF ( var, evta + evtb )
  THEN ...      !at least one of var.<9> and var.<11> is zero
  ELSE ...;     !both var.<9> and var.<11> are one
```

_ANYON

_ANYON is a Boolean define statement that is TRUE if any one-bit of *bit-mask* is on in *int-exp*. TRUE is nonzero, not necessarily -1.

The _ANYON function is the same as the _ON function. It is more descriptive to use _ANYON when testing more than one bit.

```
_ANYON ( int-exp , bit-mask )
```

int-exp input

INT:value

is the INT expression compared with *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-exp* to test.

Example

The following example tests if at least one of bits 9 and 11 in var are on:

```
INT var;
LITERAL evta = %20;   !evta.<11> on
LITERAL evtb = %100;  !evtb.<9> on

IF _ANYON ( var, evta + evtb )
  THEN ...      !at least one of var.<9> and var.<11> is one
  ELSE ...;     !both var.<9> and var.<11> are zero
```

_ANYON^TURNOFF

If any one-bit of *bit-mask* is on in *int-var*, _ANYON^TURNOFF returns TRUE and turns off any one-bits in *int-var* that are on in *bit-mask*. TRUE is nonzero, not necessarily -1.

```
_ANYON^TURNOFF ( int-var , bit-mask )
```

int-var input/output

INT:ref

is the variable compared with *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-var* to turn off.

_APPEND^OUTPUT

_APPEND^OUTPUT appends text and other variable-length items to an output list member. The maximum allowed length of a character string (ZDSN-MAX-TEXT) is 75 characters.

Text and other variable-length items can only be appended to the frame output list.

```
error := _APPEND^OUTPUT ( output-list-member
                         ,type
                         ,[ header ]
                         ,[ header-len ]
                         ,[ body ]
                         ,[ body-len ] );
```

error returned value

INT

is a ZDSN^ERR value indicating the outcome of the call. See Appendix B, “DSNM Error Codes,” for error code definitions.

output-list-member input

INT .EXT

is the output list member to which an item is appended.

type input

INT:value

is one of the following ZDSN^VTY codes describing the appended item:

ZDSN^VTY^RESULTTEXT	One line of additional subsystem and object-specific text that further explains the value in the output object’s result code field. For the STATUS command, describes subsystem state of object. Provides brief error description if error occurs.
ZDSN^VTY^TEXT	Additional lines of explanatory text for the STATUS (DETAIL), INFO, and STATISTICS commands.
ZDSN^VTY^ERRTEXT	Text that describes error conditions for detailed error requests.
ZDSN^VTY^NONTEXT	Indicates that it is not possible to scan the header, and the header length is required.
ZDSN^VTY^COUNTERS	State summary counters for the AGGREGATE command.

header input

STRING .EXT

is the appended item's header. If *header* is terminated with a null, then *header-len* is not required.

header-len input

INT:value

is the length, in bytes, of the appended item's header. Required only if *header* does not terminate with a null byte.

body input

STRING .EXT

is the item to be appended to the output list member. If *body* terminates with a null byte, *body-len* is not required. *body* should be appropriate to the value represented.

body-len input

INT:value

is the length of the appended item, in bytes. Required only if *body* does not terminate with a null byte.

Considerations

- *header* and *body* together represent a "keyword : value" pair. *body* with no *header* is any other line of text. Text items appended to an object are displayed with the object in the following form:

  object [*result-code*] , *body* (ZDSN^VTY^RESULTTEXT)

  [*header*] : *body* (ZDSN^VTY^TEXT)

  [*header*] : *body* (ZDSN^VTY^ERRTEXT)

  counter-values (ZDSN^VTY^COUNTERS)

- A response may have several text and/or error text lines (see individual command requirements in Section 4, "DSNM Command Requirements").

- A type ZDSN^VTY^COUNTERS structure is described by ZDSN^DDL^COUNTERS^DEF and contains the number of objects of the Z^OBJTYPE in each DSNM state. The relevant counters structure fields are:

  INT(32) Z^GREEN;

  INT(32) Z^UP = Z^GREEN;

  INT(32) Z^RED;

  INT(32) Z^DOWN = Z^RED;

  INT(32) Z^YELLOW;

  INT(32) Z^PENDING = Z^YELLOW;

  INT(32) Z^UNDEFINED;

  INT(32) Z^INERROR;

  For more information, see the description of the AGGREGATE command in Section 4, “DSNM Command Requirements.”

Example

```
IF (err := _APPEND^OUTPUT (cx.outobj, ZDSN^VTY^COUNTERS,,,
                           count, $LEN (count)))
  THEN RETURN err;
```

_BITDEF

_BITDEF defines a bit within a specified range of bits. A compile error is generated if *int* falls outside of the range *min-bit* to *max-bit*, inclusive.

```
_BITDEF ( int [,max-bit ] [,min-bit ] )
```

int input

the bit position to be defined within the range *min-bit* to *max-bit*, inclusive.

max-bit input

the leftmost bit position within the word that is now considered bit position 0. The default is %100000.

min-bit input

the rightmost bit position within the range. The default is 1.

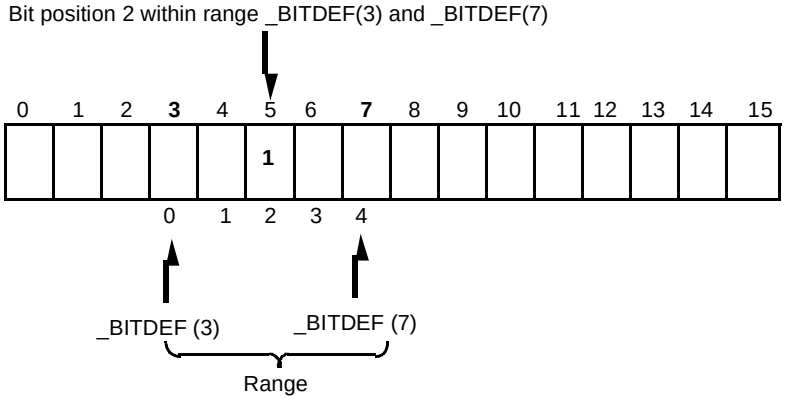
Examples

The following example assigns the value of bit position 2 within the range of bit positions 3 and 7 in a word to evtb:

```
LITERAL maxval = _BITDEF (3);                ! maxval = %10000
LITERAL minval = _BITDEF (7);                ! minval = %400
LITERAL evtb = _BITDEF (2, maxval, minval); ! evtb = %2000
```

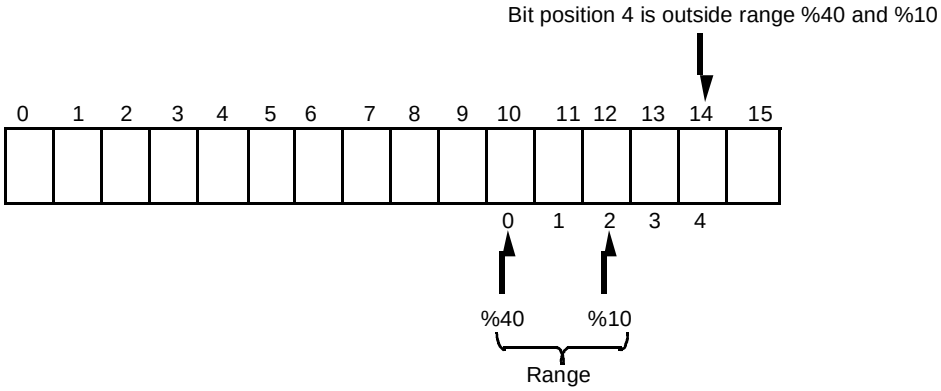
This is the same as:

```
LITERAL evtb = _BITDEF (2, %10000, %400);   ! evtb = %2000
```



The next example generates a compile error because it attempts to assign a value to a bit position outside of the designated range:

```
LITERAL z = BITDEF (4, %40, %10);
```



Bit position 4 requires a rightmost bit position range delimiter equal to or greater than %2 (_BITDEF(14)).

_CANCEL^SEND^CI

_CANCEL^SEND^CI cancels an outstanding _SEND^CI operation. The thread cancels all outstanding _SEND^CI operations when the thread terminates.

```
error := _CANCEL^SEND^CI ( [ tag ] );
```

error returned value

INT

is a file system error. File system errors are documented in the *Guardian Procedure Errors and Messages Manual*.

tag input

INT(32):value

is the tag of an outstanding _SEND^CI operation.

If *tag* is omitted, the frame selects an outstanding operation on that CI from this thread to be canceled.

Example

```
IF (error := _CANCEL^SEND^CI)
  THEN ... ;
```

_CANCEL^TIMEOUT

_CANCEL^TIMEOUT cancels an outstanding timeout set by a call to _SET^TIMEOUT.

The tag of the canceled operation is stored in the command context space and can be accessed with _LAST^TIMEOUT^TAG.

```
error := _CANCEL^TIMEOUT ( [ tag ] );
```

error returned value

INT

is a ZDSN^ERR value indicating the outcome of the call. See Appendix B, “DSNM Error Codes,” for error code definitions.

tag input

INT(32):value

is the tag of an outstanding _SET^TIMEOUT operation.

If *tag* is omitted, the frame selects an outstanding timeout from this thread to cancel, if any.

_CI^DEF

_CI^DEF is a template for a CI configuration structure, filled in by the _ADD^CI procedure.

```
_CI^DEF
```

You must declare an extended pointer to a _CI^DEF-defined CI configuration structure in globals for each CI with which your I process communicates.

_ADD^CI must be called in your _STARTUP procedure for each CI class your with which your I process communicates. (_ADD^CI allocates the memory for, fills in, and returns the address of the CI configuration structure.)

Considerations

The definition of the _CI^DEF-defined structure is:

```
DEFINITION ZDSN-DDL-PCLASS-CONFIG.
  02 Z-PCLASS               TYPE ZDSN-DDL-PCLASS.
  02 Z-PUBLIC-NAME-OCCURS   TYPE ZSPI-DDL-UINT.
  02 Z-PUBLIC-NAME          TYPE ZDSN-DDL-PARAMNAME.
  02 Z-FLAGS                TYPE ZSPI-DDL-ENUM.
  02 Z-PNAME-OCCURS         TYPE ZSPI-DDL-UINT.
  02 Z-PNAME                TYPE ZDSN-DDL-PNAME.
  02 Z-MAX-PROCESSES        TYPE ZSPI-DDL-INT.
  02 Z-OPEN-PARAMS.
    03 Z-DEFAULT-QUALIFIER TYPE ZDSN-DDL-PQUAL.
    03 Z-NOWAIT-DEPTH      TYPE ZSPI-DDL-INT.
    03 Z-OPEN-TIMEOUT      TYPE ZSPI-DDL-INT2.
  02 Z-NEWPROCESS-PARAMS.
    03 Z-OBJECT-FILE       TYPE ZDSN-DDL-OBJNAME.
    03 Z-LIBRARY-FILE      TYPE ZDSN-DDL-OBJNAME.
    03 Z-SWAPVOL           TYPE ZDSN-DDL-OBJNAME.
    03 Z-PRIORITY          TYPE ZSPI-DDL-INT.
    03 Z-DATAPAGES         TYPE ZSPI-DDL-INT.
    03 Z-NUM-CPUS          TYPE ZSPI-DDL-INT.
    03 Z-CPUS              TYPE ZSPI-DDL-INT OCCURS 16 TIMES.
    03 Z-HOMETERM          TYPE ZDSN-DDL-OBJNAME.
    03 Z-FLAGS             TYPE ZSPI-DDL-ENUM.
END
```

Example

```
< in global definitions >
INT .EXT scp (_CI^DEF);
INT .EXT snaxcdf (_SUBSYS^DEF);

STRING .scpclass[0:ZDSN^MAX^CICLASS-1] := [“SCP ”];
STRING .cdf[0:ZDSN^MAX^SUBSYS-1] := [“SNAXCDF ”];

< within _STARTUP procedure >
   .
   .
  IF _ISNULL (@scp := _ADD^CI (scpclass)) THEN
    RETURN ZDSN^ERR^INTERNAL^ERR;
  IF _ISNULL (@snaxcdf := _ADD^SUBSYS (cdf)) THEN
    RETURN ZDSN^ERR^INTERNAL^ERR;
   .
   .
< within _COMMAND^PROC procedure >
   .
   .
  CALL _OPEN^CI (scp, ... );
   .
   .
```

_CI^FILENUM

_CI^FILENUM gives the type INT file number of the CI involved with the most-recently completed communication.

```
filenumber := _CI^FILENUM ( ciid )
```

filenumber output

INT

is the INT Guardian file number of the CI involved in the most-recently completed communication.

ciid input

is the CIID structure (declared with _CI^ID) identifying an open CI.

Example

```
CALL FILEINFO (_CI^FILENUM (ciid), error);
```

_CI^ID

_CI^ID declares a structure (referred to as a “CIID” structure) in which _OPEN^CI stores information about an open CI.

```
_CI^ID ( ciid );
```

ciid user-provided identifier

is the name (a valid TAL identifier) given to the CIID structure with which an open CI can be accessed.

The CIID structure plays a role in command-thread CI communication analogous to a file number in Tandem NonStop Kernel interprocess communications. A particular instance of an open CI is identified by its *ciid* in CI communications.

The following defines extract information from the CIID structure about the communication just completed:

_CI^LASTERROR (*ciid*)	INT file-system error of last operation
_CI^REPLYLENGTH (*ciid*)	INT length of reply
_CI^REPLYADDRESS (*ciid*)	INT(32) extended address of reply
_CI^REPLYTAG (*ciid*)	INT(32) tag of last operation
_CI^FILENUM (*ciid*)	INT Guardian file number of CI

Example

The following example opens a CI:

```
_CI^ID (pm);
INT .EXT ci^config (_CI^DEF);

IF (error := _OPEN^CI (ci^config, pm)) THEN ...;
```

_CI^IDPOINTER

_CI^IDPOINTER_CI^IDPOINTER declares an extended pointer to a CIID structure.

After the thread is dispatched by the frame with _EV^IODONE, _LAST^CI^ID is a type _CI^IDPOINTER pointer, which gives access to information about the CI causing the event.

```
_CI^IDPOINTER ( ciid );
```

ciid input

declares a pointer to a CIID structure (declared with _CI^ID).

Example

The following example declares a pointer to a CIID structure:

```
_CI^IDPOINTER (pm);           !pointer to a CIID structure


IF _ON (_LAST^EVENTS, _EV^IODONE)
 THEN
   BEGIN
     @pm := _LAST^CI^ID;      !pm gets address of CIID struct
       IF _CI^LASTERROR (pm)  !check for errors
         THEN ... ;
   END;
```

_CI^LASTERROR

_CI^LASTERROR is the type INT file-system error of the last CI operation.

```
fserror := _CI^LASTERROR ( ciid )
```

fserror output

INT

is the file-system error of the last CI operation.

ciid input

is the CIID structure (declared with _CI^ID) identifying an open CI.

Considerations

If _CI^LASTERROR is not 0, the following actions occurred:

- Retriable errors were retried unsuccessfully.
- If the context flag of _SEND^CI was false (the send was context-free), an attempt was made to reestablish communication with the CI process and to send the request again.

  If communication is reestablished, the file number returned by _CI^FILENUM can be different from earlier CI communications.

  If any of these actions results in a successful communication, _CI^LASTERROR is 0; otherwise, it is the last error that occurred. File system errors are documented in the *Guardian Procedure Errors and Messages Manual*.

Example

```
_CI^IDPOINTER (pm);           !pointer to a CIID structure

IF _ON (_LAST^EVENTS, _EV^IODONE)
 THEN
   BEGIN
     @pm := _LAST^CI^ID;      !pm gets address of CIID struct
       IF _CI^LASTERROR (pm)  !check for errors
         THEN ... ;
   END;
```

_CI^REPLYADDRESS

_CI^REPLYADDRESS is the type INT(32) extended address of the reply buffer containing information read from a CI on completion of a _SEND^CI.

```
@replyaddress := _CI^REPLYADDRESS ( ciid )
```

replyaddress output

INT(32)

is the extended address of the reply buffer.

ciid input

is the CIID structure (declared with _CI^ID) identifying an open CI.

Example

In the following example, `cireply` is set to point to the reply buffer containing data returned by the most-recently completed I/O operation:

```
INT .EXT cireply;

IF _ON (_LAST^EVENTS, _EV^IODONE )
  THEN
    BEGIN
      @cireply := _CI^REPLYADDRESS (_LAST^CI^ID);
      ...
    END;
```

_CI^REPLYLENGTH

_CI^REPLYLENGTH is the type INT length of the reply buffer containing information read from a CI on completion of a _SEND^CI.

```
replylength := _CI^REPLYLENGTH ( ciid )
```

replylength output

INT

is the length of the reply buffer.

ciid input

is the CIID structure (declared with _CI^ID) identifying an open CI.

Example

In the following example, `replylen` is the length of the reply buffer containing data returned by the most-recently completed CI I/O operation:

```
INT replylen;

IF _ON (_LAST^EVENTS, _EV^IODONE)
  THEN
    BEGIN
      replylen := _CI^REPLYLEN (_LAST^CI^ID);
      ...
    END;
```

_CI^REPLYTAG

_CI^REPLYTAG is the type INT(32) tag associated with the last CI operation.

```
replytag := _CI^REPLYTAG ( ciid )
```

replytag output

INT(32)

is the tag associated with the last CI operation.

ciid input

is the CIID structure (declared with _CI^ID) identifying an open CI.

Example

In the following example, `replytag` is the tag associated with the last CI operation:

```
INT(32) replytag;

IF _ON (_LAST^EVENTS, _EV^IODONE)
  THEN
    BEGIN
      replytag := _CI^REPLYTAG (_LAST^CI^ID);
      ...
    END;
```

_CLOSE^CI

_CLOSE^CI terminates a CI communication. In addition, _CLOSE^CI cancels all outstanding I/O operations.

You must close a CI before its CIID structure (*ciid*) can be used in another _OPEN^CI operation.

```
error := _CLOSE^CI ( ciid );
```

error returned value

INT

is a file-system error. File system errors are documented in the *Guardian Procedure Errors and Messages Manual*.

ciid input

is the CIID structure (declared with _CI^ID) identifying an open CI.

Example

In the thread termination procedure:

```
      .
      .
CALL _CLOSE^CI (cx.spif);
      .
      .
```

_COMMAND^CONTEXT^HEADER

_COMMAND^CONTEXT^HEADER defines and reserves the fixed header portion of the command context space that is allocated to each thread when it is created and that persists until the thread terminates. This part of the command context space is reserved for the specific uses described in Section 3, "I Process Development Process."

```
_COMMAND^CONTEXT^HEADER;
```

Considerations

All private data used by the command thread must be defined in the command context space or members of lists. Data in global areas is shared by all active threads and may only be used as read-only data. Data in the local procedure is destroyed with each return to the frame by any of the following:

- _RETURN _RC^*xxx*
- _DISPATCH^THREAD
- _SAVE^THREAD^AND^DISPATCH
- _RESTORE^THREAD^AND^DISPATCH

The frame initializes the user-data area of the command context space to 0. Specify the length of the command context structure in your _STARTUP procedure, as in the following example:

```
INT PROC _STARTUP (context^length, input^lm^length) EXTENSIBLE;
INT .context^length, .input^lm^length;
 BEGIN
  context^length := $LEN (command^context^def);
  input^lm^length := $LEN (input^lm^def);
       .
  RETURN ZDSN^ERR^NOERR;
 END;
```

Example

The following example declares a command context structure:

```
STRUCT COMMAND^CONTEXT^DEF (*);
 BEGIN
  _COMMAND^CONTEXT^HEADER;
  <user-defined data>
    .
 END;
```

See the _FOBJECT^INIT description for another _COMMAND^CONTEXT^HEADER example.

_COMMAND^PROC

_COMMAND^PROC is the name of the first command thread procedure to be dispatched by the frame.

```
_THREAD^PROC (_COMMAND^PROC);
```

Considerations

The frame invokes the command thread as _COMMAND^PROC the first time it dispatches an instance of the thread. It is required to have one thread defined with this name in the I process.

You can change the procedure called at the next dispatch with:

- _DISPATCH^THREAD
- _PUSH^THREAD^PROCSTATE
- _POP^THREAD^PROCSTATE
- _SET^THREAD^PROC
- _SAVE^THREAD^AND^DISPATCH
- _RESTORE^THREAD^AND^DISPATCH

Note. The thread procedure called at the next dispatch is referred to as the “current” thread.

Example

```
_THREAD^PROC (_COMMAND^PROC);
  BEGIN
    < procedure body >
  _END^THREAD^PROC;
```

_COMMAND^TERMINATION^PROC

_COMMAND^TERMINATION^PROC is the name of the thread termination procedure declared with _THREAD^TERMINATION^PROC.

```
_THREAD^TERMINATION^PROC ( _COMMAND^TERMINATION^PROC);
```

Example

Use _COMMAND^TERMINATION^PROC in the following construction:

```
_THREAD^TERMINATION^PROC (_COMMAND^TERMINATION^PROC);
  BEGIN
     .
   < procedure body >
     .
     RETURN _RC^NULL;
  _END^THREAD^TERMINATION^PROC;
```

_COMPILED^IN^TESTMODE

_COMPILED^IN^TESTMODE is a literal with a value of 1, if a source file is compiled with the SETTOG 1 compiler directive; otherwise, _COMPILED^IN^TESTMODE is 0.

Use _COMPILED^IN^TESTMODE to set the value of the *testmode* parameter in your _STARTUP^MODE procedure.

```
_COMPILED^IN^TESTMODE
```

Example

```
INT PROC _STARTUP^MODE (component, testmode,
                        accept^startup^component) EXTENSIBLE;

STRING .EXT component;
INT .EXT accept^startup^component;
BEGIN
  testmode := _COMPILED^IN^TESTMODE;
  accept^startup^component := 1;
  RETURN ZDSN^ERR^NOERR;
END;
```

_DEALLOCATE^LIST

_DEALLOCATE^LIST deletes all members of a list. Memory for the list members is deallocated immediately.

```
CALL _DEALLOCATE^LIST ( list );
```

list input

is the name of a _LIST.

Example

```
CALL _DEALLOCATE^LIST (cx.xc.input);
```

_DELETE^LM

_DELETE^LM deletes a member of a list, immediately deallocates its memory, and sets the list-member pointer to null.

```
error := _DELETE^LM ( list
                     ,@list-member );
```

error returned value

INT

is a ZDSN^ERR value, indicating the outcome of the call. See Appendix B, “DSNM Error Codes,” for error code definitions.

list input

is the name of a _LIST.

list-member input/output

INT .EXT

is a pointer to the member to be deleted from *list*.

list-member must point to a current member of *list*, or results are unpredictable.

_DEPOSIT

_DEPOSIT sets selected bits in the first parameter equal to the same bits in the second parameter.

```
_DEPOSIT ( int-var
          ,int-exp
          ,bit-mask );
```

int-var input/output

INT:ref

is INT variable, the selected bits of which are set equal to the same bits in *int-exp*.

int-exp input

INT:value

is an INT expression, the selected bits to which *int-var* bits are set equal.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-var* and *int-exp* participating in the operation.

Considerations

_DEPOSIT performs a function similar to the following, except that the affected bits need not be contiguous:

```
a.<x:y> := b.<x:y>
```

Example

In the following example, bits 9 and 15 of `evta` are set equal to bits 9 and 15 of `evtb`:

```
INT evta, evtb;

_DEPOSIT (evta, evtb, %101);  !sets evta.<9> := evtb.<9>
                              !and evta.<15> := evtb.<15>.
                              !Other bits of evta unchanged.
```

_DISPATCH^THREAD

_DISPATCH^THREAD returns to the frame and causes a new dispatch. It is effective only in a thread procedure, not in an auxiliary procedure or a subprocedure.

_DISPATCH^THREAD saves no information about the procedure from which it was invoked.

This function cannot detect any failures and performs an unconditional RETURN operation.

```
_DISPATCH^THREAD ( [ @procname ]
                  ,[ state ]
                  ,[ event ] );
```

procname input

is the dispatched thread procedure. The default is the current procedure.

state input

INT: value

is an INT expression that designates the new current thread state when the thread is dispatched. The default is the current state.

event input

INT:value

is an INT expression that designates the event(s) with which the new procedure is dispatched. The default is _EV^CONTINUE.

To use _DISPATCH^THREAD with no arguments (accepting all defaults), you must use the following construction:

```
_DISPATCH^THREAD ( );
```

To immediately redispatch the current thread in the current state, use _DISPATCH^THREAD with the first two arguments blank:

```
_DISPATCH^THREAD ( , , event );
```

Example

```
_DISPATCH^THREAD (@next^proc, ,_REAL^LAST^EVENTS);

_DISPATCH^THREAD (@myproc, _ST^INITIAL, my^event);

_DISPATCH^THREAD (@myproc, my^state, my^event);
```

_DSNMCONF^PARAMS

_DSNMCONF^PARAMS is a global structure, defined and stored in the I process globals. The frame retrieves these parameters as part of its startup function.

```
STRUCT _DSNMCONF^PARAMS (ZDSN^DDL^DSNMCONF^PARAMS^DEF);
```

Considerations

The contents of the _DSNMCONF^PARAMS structure is as follows:

```
DEFINITION ZDSN-DDL-DSNMCONF-PARAMS.
  02 Z-DSNM-MANAGER-OCCURS   TYPE ZSPI-DDL-UINT.
  02 Z-DSNM-MANAGER          TYPE ZDSN-DDL-MANAGER.
  02 Z-SWAPVOL-OCCURS        TYPE ZSPI-DDL-UINT.
  02 Z-SWAPVOL               TYPE ZDSN-DDL-OBJNAME.
  02 Z-SEGPAGES              TYPE ZSPI-DDL-INT2.
  02 Z-SEGEXT.
    03 Z-PRIMARY             TYPE ZSPI-DDL-INT.
    03 Z-SECONDARY           TYPE ZSPI-DDL-INT.
  02 Z-OBJECT-DB-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-OBJECT-DB             TYPE ZDSN-DDL-OBJNAME.
  02 Z-OBJECT-MONITOR-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-OBJECT-MONITOR        TYPE ZDSN-DDL-PNAME.
  02 Z-OBJECT-DB-INTERFACE-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-OBJECT-DB-INTERFACE   TYPE ZDSN-DDL-PNAME.
  02 Z-MAX-OPENERS           TYPE ZSPI-DDL-INT.
  02 Z-EMS-COLLECTOR-OCCURS  TYPE ZSPI-DDL-UINT.
  02 Z-EMS-COLLECTOR         TYPE ZDSN-DDL-PNAME.
  02 Z-SECPARAMS             TYPE ZSPI-DDL-UINT.
END
```

_EMPTY^LIST

_EMPTY^LIST is a Boolean define statement that is TRUE if *list* has no members. TRUE is nonzero, not necessarily -1.

```
_EMPTY^LIST ( list )
```

list input

is the name of a _LIST .

Example

The following example tests if `list` is empty:

```
_LIST (list);

IF _EMPTY^LIST (list)
  THEN ...;
```

_EMS^EVENT^CRITICAL

_EMS^EVENT^CRITICAL is a value used in the _REPORT^INTERNAL^ERROR and _REPORT^STARTUP^ERROR procedures, indicating that the event being logged to EMS is critical but not fatal.

```
_EMS^EVENT^CRITICAL
```

_EMS^EVENT^FATAL

_EMS^EVENT^FATAL is a value used in the _REPORT^INTERNAL^ERROR and _REPORT^STARTUP^ERROR procedures, indicating that the event being logged to EMS is fatal.

```
_EMS^EVENT^FATAL
```

_EMS^EVENT^INFO

_EMS^EVENT^INFO is a value used in the _REPORT^INTERNAL^ERROR and _REPORT^STARTUP^ERROR procedures, indicating that the event being logged to EMS is a non-fatal informative message.

```
_EMS^EVENT^INFO
```

_END^THREAD^PROC

_END^THREAD^PROC ends a thread procedure definition. Any procedure that can be dispatched as part of a thread must be declared with _THREAD^PROC and _END^THREAD^PROC.

_END^THREAD^PROC issues RETURN _RC^WAIT.

```
_END^THREAD^PROC;
```

Example

```
_THREAD^PROC (procname );
  BEGIN
   < procedure body >
  _END^THREAD^PROC;
```

_END^THREAD^TERMINATION^PROC

_END^THREAD^TERMINATION^PROC ends a thread termination procedure definition and issues RETURN _RC^NULL.

The thread termination procedure declared with _THREAD^TERMINATION^PROC must end with _END^THREAD^TERMINATION^PROC.

```
_END^THREAD^TERMINATION^PROC;
```

Example

Use _END^THREAD^TERMINATION^PROC in the following construction:

```
_THREAD^TERMINATION^PROC (_COMMAND^TERMINATION^PROC);
  BEGIN
   < procedure body >
  _END^THREAD^TERMINATION^PROC;
```

_EV^CANCEL

_EV^CANCEL is generated by the frame when it receives a command cancellation request.

The command thread can be dispatched with _EV^CANCEL after any return to the frame; the operation should be terminated immediately.

```
_EV^CANCEL
```

_EV^CANCEL should be handled like an error: you must perform clean-up operations, such as freeing resources, returning the CI to a reasonable state, and so on. Since _EV^CANCEL is a normal thread termination, the command thread should return to the frame with an _RC^STOP return code.

_EV^CONTINUE

EV^CONTINUE is generated by the frame when the thread returns with _RC^WAIT, and there is no outstanding request to complete.

```
_EV^CONTINUE
```

_EV^IODONE

_EV^IODONE is generated by the frame when I/O initiated by _SEND^CI completes. (*cmd^context*._LAST^CI^ID gets the address of the CIID structure of the completed operation.)

```
_EV^IODONE
```

_EV^STARTUP

_EV^STARTUP is generated by the frame on its initial dispatch of a command thread.

```
_EV^STARTUP
```

_EV^TIMEOUT

_EV^TIMEOUT is generated by the frame when a timeout interval set by a call to _SET^TIMEOUT elapses. (*cmd^context*._LAST^TIMEOUT^TAG gets the tag of the elapsed timeout.)

```
_EV^TIMEOUT
```

Note. The thread may simulate any frame event by signaling it with _SIGNAL^EVENT.

_EXTRACT

_EXTRACT returns the value of those bits of *bit-mask* that are on in *int-exp*. _EXTRACT performs a function similar to the TAL bit-extraction operation, except the extracted bits need not be contiguous, nor are they shifted to the right.

```
_EXTRACT ( int-exp , bit-mask );
```

int-exp input

INT:value

is an INT expression from which bits are extracted, according to the one-bits in *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-exp* to extract.

Example

```
LITERAL error^bits = %B1101;
INT sense^code, errors;

      !Suppose sense^code = %B111000

errors := _EXTRACT (sense^code, error^bits); !errors = %B1000
```

_FIRST^LM

_FIRST^LM returns the address of the first member of a list. _NULL is returned if the list is empty.

```
@first-list-member := _FIRST^LM ( list );
```

first-list-member returned value

INT .EXT

is the address of the first member of *list*.

list input

is the name of a _LIST.

Example

```
_LIST (outlist);
INT .EXT list^member (list^member^def); !declare extended
                                       !pointer to list
                                       !member structure

@list^member := _FIRST^LM (outlist);    !get address of first
                                       !member
```

FOBJECT

Every subsystem object processed by DSNM is defined by the contents of a ZDSN^DDL^FOBJECT^DEF structure, known as a "formatted object." The _INPUT^LM^HEADER and _OUTPUT^LM^HEADER defines assign the name FOBJ to the formatted object structure portions of input and output list members.

It is important that every object processed by the command thread be represented in an FOBJECT structure, properly initialized with the _FOBJECT^INIT procedure.

The FOBJECT structure contains fields used directly by the command thread; it also contains internal fields used by the I process frame and libraries.

Example

In the following example, information about the subordinate of an input object is entered into an initialized output object for release to the frame:

```
STRUCT input^lm^def (*);
  BEGIN
    _INPUT^LM^HEADER;
   ...
  END;

STRUCT output^lm^def (*);
  BEGIN
    _OUTPUT^LM^HEADER;
   ...
  END;

STRUCT command^context^def (*);
  BEGIN
    _COMMAND^CONTEXT^HEADER;
    INT .EXT inobj (input^lm^def);
    INT .EXT outobj (output^lm^def);
  END;

!Thread proc locals!

INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT in (input^def) := @cx._INPUT;
INT .EXT out (output^def) := @cx._OUTPUT;
       .
       .
```

```
! Create output list member

IF _ISNULL (@cx.outobj := _PUT^LM (out.OBJECTLIST,,
                                   $LEN (cx.outobj)))
  THEN ... < out of available memory > ;
       .
       .
   IF (error := _FOBJECT^INIT (cx.outobj.FOBJ,,
                               cx.inobj.FOBJ))
    THEN ... < error exit > ;
   cx.outobj.FOBJ.Z^RESULT := < status of subordinate >;
   cx.outobj.FOBJ.Z^OBJTYPE ':=' < type of subordinate >;
   cx.outobj.FOBJ.Z^OBJNAME ':=' < name of subordinate >;
   _RELEASE^OUTPUT (cx.outobj);
       .
       .
  END;
```

_FOBJECT^INIT

_FOBJECT^INIT initializes a new FOBJECT structure and determines required fields from its source FOBJECT structure. _FOBJECT^INIT does not allocate memory; memory for the new formatted object must be allocated previously.

```
error := _FOBJECT^INIT ( new-fobject
                        ,[ same-fobject ]
                        ,[ parent-fobject ]);
```

error returned value

INT

is a ZDSN^ERR value, indicating the outcome of the call. See Appendix B, "DSNM Error Codes," for error code definitions.

new-fobject user-provided identifier

is the name (a valid TAL identifier) of the new FOBJECT structure to initialize.

same-fobject input

is the name of an FOBJECT structure that contains the same object information as *new-fobject*. In this case, *new-fobject* is identical to its source object.

parent-fobject input

is the name of an FOBJECT structure from which *new-fobject* is derived when processing a hierarchy modifier or expanding a "*" object name.

Either *same-fobject* or *parent-fobject* must be supplied in the call, but not both. In both cases, all required internal information is entered in the *new-fobject* structure.

- Use the *same-fobject* argument if the new FOBJECT structure is to define the same object as an existing FOBJECT structure. The new object is the same if it has the same subsystem, object type, name, and manager. In this case, use the following syntax to initialize the new FOBJECT structure:

  error := _FOBJECT^INIT (*new-fobject*, *same-fobject*);

  The following fields from the source FOBJECT structure are copied to *new-fobject* when the *same-fobject* argument is supplied:

  Z^SUBSYS
  Z^OBJTYPE
  Z^OBJNAME^OCCURS
  Z^OBJNAME
  Z^MANAGER^OCCURS
  Z^MANAGER

- Use the *parent-fobject* argument if the new FOBJECT structure is to define a different object from any previously initialized FOBJECT structure. Specify the new object's parent in the name hierarchy as described above as the *parent-fobject*. The new object is different if it differs in any of subsystem, object type, name, or manager from its name parent (the name from which the new object was derived by expanding a "*" or through the subsystem hierarchy). In this case, use the following syntax to initialize the new FOBJECT structure:

  ```
  error := _FOBJECT^INIT (new-fobject,,parent-fobject);
  ```

  The following fields from the source FOBJECT structure are copied to *new-fobject* when the *parent-fobject* argument is supplied:

  Z^SUBSYS
  Z^MANAGER^OCCURS
  Z^MANAGER

  Z^OBJTYPE, Z^OBJNAME, and Z^OBJNAME^OCCURS are set to null values (0 or blanks, as appropriate). You must supply values for Z^OBJTYPE and Z^OBJNAME. Suppling a value for Z^OBJNAME^OCCURS is optional.

  In both cases, all required internal information is entered into the *new-fobject* structure.

Considerations

Parent means the parent of the new object in a name hierarchy, which includes the subsystem hierarchy and a "*" object name, if supported by your I process. You can produce new objects from objects on the input list in two ways:

1. The input object is a subsystem object, and new object names are subordinate objects produced by processing a hierarchy modifier (HMOD).
2. The input object is a "*," and new object names are produced by expanding the "*."

In either case, the input object is the parent of the new object in the name hierarchy (which includes the subsystem hierarchy).

Note. Outside the I process, there are higher levels possible in the name hierarchy: DNS groups (possibly nested) and composites.

Example

In the following example, each input object and its hierarchical subordinates appear in the output for a STATUS command:

```
STRUCT input^lm^def (*);
  BEGIN
    _INPUT^LM^HEADER;
   ...
  END;

STRUCT output^lm^def (*);
  BEGIN
    _OUTPUT^LM^HEADER;
   ...
  END;

STRUCT command^context^def (*);
  BEGIN
    _COMMAND^CONTEXT^HEADER;
    INT .EXT inobj (input^lm^def);
    INT .EXT outobj (output^lm^def);
  END;

!Thread proc locals!

INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT in (input^def) := @cx._INPUT;
INT .EXT out (output^def) := @cx._OUTPUT;
    ...

! Get the next input object

IF _ISNULL(@cx.inobj := _GET^LM (in.OBJECTLIST))
  THEN RETURN _RC^STOP;

!Create output list member

IF _ISNULL (@cx.outobj := _PUT^LM (out.OBJECTLIST,,
                                   $LEN (cx.outobj)))
  THEN ... < out of available memory > ;

! Now cx.inobj.fobj and cx.outobj.fobj are the current
! input and output list members.  Since the output object
! will be the same as the input object, use the same-fobject
! parameter:

IF (error := _FOBJECT^INIT (cx.outobj.FOBJ,
                            cx.inobj.FOBJ))
  THEN ... < error exit > ;

! Now cx.inobj.fobj and cx.outobj.fobj are the current
! input and output objects.

   ! Send to CI, determine status of input object and its
   ! subordinates.
```

```
   ! Use state variables to return to this point after the
   ! _EV^IODONE event occurs

cx.outobj.FOBJ.Z^RESULT := < status of input object >;

! Since this completes the current output object, release it

_RELEASE^OUTPUT (cx.outobj);

! Enter subordinates and their status into output list
! (Assuming one CI communication returns all subordinates)

WHILE < more subordinate objects >
 DO
  BEGIN
   IF _ISNULL (@cx.outobj := _PUT^LM (out.OBJECTLIST,,
                                      $LEN (cx.outobj)))
      THEN ... < out of available memory > ;
  ! Next output object

  ! Since the output object is not the same as the input
  ! object, use the parent-fobject parameter:

   IF (error := _FOBJECT^INIT (cx.outobj.FOBJ,,
                               cx.inobj.FOBJ))
    THEN ... < error exit > ;

   cx.outobj.FOBJ.Z^RESULT := < status of subordinate >;
   cx.outobj.FOBJ.Z^OBJTYPE ':=' < type of subordinate >;
   cx.outobj.FOBJ.Z^OBJNAME ':=' < name of subordinate >;

   _RELEASE^OUTPUT (cx.outobj);
   ...
  END;
```

See Section 4, “DSNM Command Requirements,” for more information about FOBJECT fields.

_GET^LM

_GET^LM removes the current first member (the earliest member put on the list) from a list and returns its address. If the list is empty, _GET^LM returns _NULL.

```
@list-member := _GET^LM ( list
                        ,[ length ] );
```

list-member returned value

INT .EXT

is the address of the removed member.

list input

is the name of a _LIST.

length output

INT:ref

returns the length of the removed member, in bytes.

Considerations

- Removing a member with _GET^LM does not immediately deallocate memory. The removed member's memory remains allocated and its contents useable until the next successive member is removed from the same end of the list, or a new member is added to the same end of the list.
- The removed member does not participate in list scans with _SUCCESSOR^LM or _PREDECESSOR^LM.
- Normally, a list is processed either by _PUT^LM plus _GET^LM or by _PUSH^LM plus _POP^LM, but not both.
- _UNGET^LM replaces the last list member removed by _GET^LM.

Example

In the following example, the first member of `outlist` is removed and `list^member` is set to point to it:

```
_LIST (outlist);
INT .EXT list^member (list^member^def);
INT length;

IF _ISNULL (@list^member := _GET^LM (outlist, length))
  THEN <empty list> ;
```

See the _FOBJECT^INIT description for another of example of _GET^LM.

_GET^PARAM

_GET^PARAM retrieves one instance of a DSNM configuration parameter that is not part of the standard set stored in the _DSNMCONF^PARAMS structure.

```
error := _GET^PARAM ( paramscope
                    , paramtype
                    ,[ subsys ]
                    ,[ class ]
                    ,[ component ]
                    , paramname
                    , paramvalue:maxlen
                    ,[ len ]
                    ,[ error-filename ] );
```

error returned value

INT

is a ZDSN^ERR or Guardian error. See Appendix B, “DSNM Error Codes,” for ZDSN^ERR error code definitions. Refer to the *Guardian Procedure Errors and Messages Manual* for Guardian error descriptions.

paramscope input

INT

indicates whether the parameter is local or global:

_LOCAL^PARAM	Local parameters consist of a single value (for example, SWAPVOL) obtained from one source: a DSNMCONF file or the startup message.
_GLOBAL^PARAM	Global parameters consist of multiple values (for example, command server SYSTEM parameters) from all sources in which instances of the parameter are found.

paramtype input

INT

indicates how restrictive the search criteria is:

_COMPONENT^PARAM	Component parameters are instances of a parameter, specific to this component and class.
_CLASS^PARAM	Class parameters are instances of a parameter, specific to this class. If *component* is blank, it is specific to the class as a whole.
_GENERAL^PARAM	General parameters are any instance of this parameter. It may be for this component, for the class as a whole (if *component* is blank), or for any class (if both *class* and *component* are blank).

subsys input

STRING .EXT ! ZDSN^DDL^SUBSYS^DEF !

is the name of the subsystem whose associated parameter values are retrieved. A blank subsystem name (all spaces) is valid; the default is “DSNM ”.

class input

STRING .EXT ! ZDSN^DDL^CLASS^DEF !

is the name of the class whose associated parameter values are retrieved. A blank class name (all spaces) is valid; if omitted, the caller’s class name is used.

component input

STRING .EXT ! ZDSN^DDL^COMPONENT^DEF !

is the name of the component whose associated parameter values are retrieved. A blank component name (all spaces) is valid; if omitted, the caller’s component name (specified by the COMPONENT parameter in your _STARTUP^MODE procedure or obtained from the process startup message) is used.

paramname input

STRING .EXT ! ZDSN^DDL^PARAMNAME^DEF !

is the name of the parameter, left-justified, blank-filled, whose value you want returned.

paramvalue output

STRING .EXT

if *error* = 0, contains the parameter value; otherwise, is undefined.

maxlen input

INT

is the maximum length returned in *paramvalue*, in bytes.

len output

INT

is the actual length of the value returned in *paramvalue*, in bytes. If *len* < *maxlen*, the remainder of *paramvalue* is blank-filled.

error-filename output

STRING .EXT ! ZDSN^DDL^OBJNAME^DEF !

is the name of the configuration file associated with the returned *error* value.

_GET^PROCESS^PARAM

_GET^PROCESS^PARAM retrieves process startup parameters not part of the standard set stored in the _PROCESS^PARAMS structure.

```
error := _GET^PROCESS^PARAM ( paramname
                            , paramvalue:maxlen
                            ,[ len ] );
```

error returned value

INT

is a ZDSN^ERR or Guardian error. See Appendix B, “DSNM Error Codes,” for ZDSN^ERR error code definitions. Refer to the *Guardian Procedure Errors and Messages Manual* for Guardian error descriptions.

paramname input

STRING .EXT ! ZDSN^DDL^PARAMNAME^DEF !

is the name of the parameter, left-justified, blank-filled, whose value you want returned.

paramvalue output

STRING .EXT

if *error* = 0, contains the parameter value; otherwise, is undefined.

maxlen input

INT

is the maximum length returned in *paramvalue*, in bytes.

len output

INT

is the actual length of the value returned in *paramvalue*, in bytes.
If *len* < *maxlen*, the remainder of *paramvalue* is blank-filled.

_INITIALIZE^LIST

_INITIALIZE^LIST sets a list structure to nulls. (Lists defined in the thread context do not have to be initialized with _INITIALIZE^LIST.)

_INITIALIZE^LIST sets the list header structure to nulls. Do not use it to deallocate members of a list (see the _DEALLOCATE^LIST description).

```
CALL _INITIALIZE^LIST ( list );
```

list input

is the name of a _LIST.

_INPUT

_INPUT is the name assigned to the _INPUT^DEF structure template within the command context area (where the frame places the command components). The _COMMAND^CONTEXT^HEADER define assigns the _INPUT name.

Example

The following example of a local data definition gives a thread procedure access to the input area:

```
INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT in (_INPUT^DEF) := @cx._INPUT;
```

See the _FOBJECT^INIT description for another _INPUT example.

_INPUT^DEF

_INPUT^DEF is a structure template into which the frame places the action and command modifiers to be passed to the thread.

```
STRUCT _INPUT^DEF (*);
  BEGIN
   _LIST (OBJECTLIST);
   INT action;
   STRUCT mod (zdsn^mod^def);
END;
```

Example

The following example of a local data definition gives a thread procedure access to the input area:

```
INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT in (_INPUT^DEF) := @cx._INPUT;
```

See the _FOBJECT^INIT description for another _INPUT^DEF example.

_INPUT^LM^HEADER

_INPUT^LM^HEADER describes the first part of the user-defined input list member structure. It is required as part of the input list member definition.

_INPUT^LM^HEADER generates a formatted object structure (ZDSN^DDL^FOBJECT^DEF); it identifies this object structure as FOBJ and identifies other fields for the frame.

```
_INPUT^LM^HEADER;
```

Considerations

The following FOBJECT fields are filled by the frame for each object in the input list:

Z^HMOD	Is the hierarchy modifier. If present, it overrides the hierarchy value in _INPUT.MOD.Z^HMOD for this object only.
Z^SUBSYS	Is the name of the subsystem.
Z^OBJTYPE	Is the object type.
Z^OBJNAME^OCCURS	Is the length of the object name.
Z^OBJNAME	Is the object name.
Z^MANAGER^OCCURS	Is the length of the manager name.
Z^MANAGER	Is the name of the manager, if any.

Other FOBJECT fields and other data items generated by _INPUT^LM^HEADER are reserved for use by the frame. See Section 4, “DSNM Command Requirements,” for more information about FOBJECT fields.

The user-data area of each input list member following the _INPUT^LM^HEADER portion is for the command thread’s use and is initialized to 0 by the frame. Specify the length of the input list member structure in your _STARTUP procedure, as in the following example:

```
INT PROC _STARTUP (context^length, input^lm^length) EXTENSIBLE;
INT .context^length, .input^lm^length;
 BEGIN
  context^length := $LEN (command^context^def);
  input^lm^length := $LEN (input^lm^def);
       .
       .
  RETURN ZDSN^ERR^NOERR;
 END;
```

Example

The following is an example of an input list member structure declaration:

```
STRUCT input^list^member^def (*);
  BEGIN
    _INPUT^LM^HEADER;
   < user-definitions >
   ...
  END;
```

See the _FOBJECT^INIT description for another _INPUT^LM^HEADER example.

_ISNULL

_ISNULL is a Boolean define statement that is TRUE if *address* is a null extended memory pointer. TRUE is nonzero, not necessarily -1.

Always use either _ISNULL or _NOTNULL to test a pointer rather than comparing it to the library literal _NULL. (There are multiple internal values that are accepted as equivalent to _NULL.)

```
_ISNULL ( address )
```

address input

INT(32):value

is the extended address being tested.

Example

Suppose `command^context^def` describing the command context area contains the following:

```
INT. EXT next^in^lm (input^lm^def);
```

The following example removes objects from the input list until the list is empty:

```
INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
     .
     .
     .
IF _ISNULL (@cx.next^in^lm := _GET^LM (cx._INPUT.OBJECTLIST))
  THEN ... < out of input objects > ;
```

See the _FOBJECT^INIT description for another _ISNULL example.

_JOIN^LIST

_JOIN^LIST appends all members of a source list to a destination list.

Data is not moved in memory; the source list is empty afterwards.

```
error := _JOIN^LIST ( dest-list
                     ,source-list );
```

error returned value

INT

is a ZDSN^ERR value indicating the outcome of the call. See Appendix B, "DSNM Error Codes," for error code definitions.

dest-list input/output

is the name of the destination _LIST to which the members of *source-list* are appended.

source-list input/output

is the name of the source _LIST whose members are appended to *dest-list*.

Example

The following example appends all the members of `worklist` to `outlist`:

```
_LIST (outlist);
_LIST (worklist);
        .
        .
        .
IF ( error := _JOIN^LIST (outlist, worklist))
  THEN ... <error> ;
```

KDSNDEFS

KDSNDEFS is the source file for all I process development definitions and declarations. Your I process program code should have the following source structure:

```
? < user compiler directives >

? SOURCE KDSNDEFS (IPROCESS^DEFINITIONS)

BLOCK PRIVATE;

        < user-defined globals >

END BLOCK;

? SOURCE KDSNDEFS (IPROCESS^GLOBALS)

       < user external procedure declarations >

? SOURCE KDSNDEFS (IPROCESS^EXTDECS)

        _STARTUP^MODE procedure

        _STARTUP procedure

        _COMMAND^PROC procedure

        < other command thread procedures >

        _COMMAND^TERMINATION^PROC procedure
```

_LAST^CI^ID

_LAST^CI^ID is a command context field that points to the CIID structure from the last _EV^IODONE event. Its type is _CI^IDPOINTER.

```
command-context._LAST^CI^ID
```

Example

In the following example, `mgr` is set to point to the CIID involved in the most-recently completed CI communication:

```
_CI^IDPOINTER (mgr);
int .ext cx(command^context^def) = _THREAD^CONTEXT^ADDRESS;

IF _ON (_LAST^EVENTS, _EV^IODONE)
 THEN
   BEGIN
     @mgr := cx._LAST^CI^ID;
       IF _CI^LASTERROR (mgr)  !check for errors
         THEN ... ;
   END;
```

_LAST^EVENTS

_LAST^EVENTS is set each time a command thread is dispatched to contain the event(s) that caused the dispatch. Each bit represents a different event.

```
_LAST^EVENTS
```

Considerations

- _LAST^EVENTS is an INT global variable that can be tested and altered; _REAL^LAST^EVENTS, which is also set to the current event(s) at each dispatch, can only be tested.
- The following events are generated by the frame:

_EV^CANCEL	When the frame receives a command cancellation request.
_EV^CONTINUE	When the thread returns with an _RC^WAIT and there is no outstanding I/O or timeout.
_EV^IODONE	When an I/O initiated by a _SEND^CI request completes.
_EV^STARTUP	On the frame's initial dispatch of the thread.
_EV^TIMEOUT	When a timeout interval set by a call to _SET^TIMEOUT elapses.

- Only one frame event at a time occurs with one dispatch per event, so only one bit of _LAST^EVENTS is ever on for a frame event.
- The thread may generate multiple, simultaneous events with _SIGNAL^EVENT. All events signaled by the thread before _RC^WAIT appear together in _LAST^EVENTS at the next thread dispatch. In this case, no frame events can appear.

Examples

The following example tests if the _EV^CANCEL bit is on in _LAST^EVENTS:

```
IF _ON (_LAST^EVENTS, _EV^CANCEL)
  THEN ... ;
```

In this example, the contents of _LAST^EVENTS are altered to reflect _EV^IODONE instead of _EV^TIMEOUT:

```
_TURNOFF (_LAST^EVENTS, _EV^TMEOUT);
_TURNON (_LAST^EVENTS, _EV^IODONE);
```

See the _SIGNAL^TIMEOUT description for another example of _LAST^EVENTS.

_LAST^LM

_LAST^LM returns the address of the last (most recent) member of a list. If a list is empty, _NULL is returned.

```
@last-list-member := _LAST^LM ( list );
```

last-list-member returned value

INT .EXT

is the address of the last member of *list*.

list input

is the name of a _LIST.

Example

```
INT .EXT cx(command^context^def) := _THREAD^CONTEXT^ADDRESS;
_LISTPOINTER (outlist) := @cx.OUTPUT.OBJECTLIST;
INT .EXT out^lm (output^lm^def);   !Declare extended pointer
                                   !to list member structure

IF _ISNULL (@out^lm := _LAST^LM (outlist))
  THEN ... < list empty > ;
```

_LAST^TIMEOUT^TAG

_LAST^TIMEOUT^TAG is a command context field, set to the INT(32) timeout tag associated with the _SET^TIMEOUT request, and completed by the last _EV^TIMEOUT event. It is convenient to use the address of a list member as a timeout tag to hold information about the purpose of the timeout, as illustrated in the example.

```
command-context._LAST^TIMEOUT^TAG
```

Example

```
STRUCT time^info^def (*);
  BEGIN
    .
    .
  END;

INT .EXT time^info (time^info^def);

IF _ISNULL(@time^info := _PUT^LM (cx.worklist,,$LEN(time^info)))
  THEN ... < out of memory > ;

     < fill in time^info data >

CALL _SET^TIMEOUT (time, @time^info);
RETURN _RC^WAIT;          !Wait for _EV^TIMEOUT
   .
   .
   .
IF _ON (_LAST^EVENTS, _EV^TIMEOUT)
  THEN
    BEGIN
      @time^info := cx._LAST^TIMEOUT^TAG;
        < process time^info data >
      CALL _DELETE^LM (cx.worklist, @time^info);
    END;
```

_LIST

_LIST declares a list structure.

```
_LIST ( list );
```

list user-provided identifier

is the name (a valid TAL identifier) of the structure that _LIST declares.

Considerations

- The data structure known as a “list” is the basis for the I process program-development software memory-management facility. A list consists of the structure declared with _LIST and the list members.
  - The list structure holds control information, used by the list library procedures. Its size and structure are fixed.
  - A list member is a block of memory, the size and description of which are determined when the member is created with _PUT^LM or _PUSH^LM.
- The following procedures and defines use `list` to extract information about and perform operations on list members:

  _DEALLOCATE^LIST (*list*)
  _DELETE^LM (*list, @list-member*)
  _EMPTY^LIST (*list*)
  _FIRST^LM (*list*)
  _GET^LM (*list*, [*length*])
  _INITIALIZE^LIST (*list*)
  _JOIN^LIST (*source-list*, *dest-list*)
  _LAST^LM (*list*)
  _MEMBERSOF^LIST (*list*)
  _POP^LM (*list*, [*length*])
  _PREDECESSOR^LM (*list*, *list-member*)
  _PUSH^LM (*list*, [*length*], *initlength* ,[*initdata*])
  _PUT^LM (*list*, [*length*], *initlength*, [*initdata*])
  _SUCCESSOR^LM (*list*, *list-member*)
  _UNGET^LM (*list*, *list-member*)
  _UNPOP^LM (*list*, *list-member*)

Example

See the example for _FOBJECT^INIT for list and list member declarations.

_LISTPOINTER

_LISTPOINTER declares an extended pointer to a _LIST structure.

Once a list pointer is initialized with a list address, it can be used anywhere _LIST is used.

```
_LISTPOINTER ( list );
```

list user-provided identifier

is the name (a valid TAL identifier) of the list pointer.

Example

```
INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
_LISTPOINTER (outlist) := @cx._OUTPUT.OBJECTLIST;
INT .EXT out^lm (output^lm^def);
   .
   .
   .
IF _ISNULL (@out^lm := _PUT^LM (outlist,,$LEN (out^lm)))
  THEN ... < out of memory > ;
   .
   .
```

_MEMBERSOF^LIST

_MEMBERSOF^LIST is the type INT(32) number of members currently in a list.

```
_MEMBERSOF^LIST ( list )
```

list input

is the name of a _LIST.

Example

In the following example, `num^members` is the number of members in `inlist`:

```
_LIST (inlist);
INT(32) num^members;

num^members := _MEMBERSOF^LIST (inlist);
```

_MOVE^LIST

_MOVE^LIST moves all members of a source list to the destination list. After the operation, the source list is empty.

```
CALL _MOVE^LIST ( dest-list, source-list )
```

dest-list input/output

is the name of the destination _LIST to which the members of *source-list* are moved.

list input/output

is the name of the source _LIST whose members are moved to the *dest-list*.

Considerations

dest-list should not have any members prior to the operation. If it does, these members will not be accessible later on, as the pointer to them will be pointing to the source members after the operation.

If *source-list* is initially empty, then, prior to the operation, it should be properly initialized: for example, with _NULL^LIST.

Example

The following example moves all members of `worklist` to `outlist`:

```
_LIST (outlist);
_LIST (worklist);

_MOVE^LIST(outlist, worklist);
```

_NOTNULL

_NOTNULL is a Boolean define statement that is TRUE if address is a nonnull extended memory pointer. TRUE is nonzero, not necessarily -1.

Always use either _NOTNULL or _ISNULL to test a pointer rather than comparing it to the library literal _NULL. (There are multiple internal values that are accepted as equivalent to _NULL.)

```
_NOTNULL ( address )
```

address input

INT(32)

is the tested extended address.

Example

The following example scans a list forward:

```
_LIST (list);
INT .EXT lm (list^member^def); !extended pointer to
                               !list member structure
@lm := _NULL;
WHILE _NOTNULL (@lm := _SUCCESSOR^LM (list,lm)) DO
  BEGIN
  ...! while pointer is not null there are more members on
     ! the list
  END;
```

_NULL

_NULL is an INT(32) literal, defining a null value for an extended memory pointer.

Always use _NULL for a null pointer value. Never test a pointer for null by comparing it to _NULL; always use _ISNULL or _NOTNULL for such tests. The I process program-development libraries use a range of null values. _NULL is guaranteed to be in the range, but is not the only possible null pointer value.

A pointer set to _NULL causes an address trap, if used, to access memory.

```
_NULL
```

Example

```
_LIST (list);
INT .EXT lm (list^member^def); !extended pointer to
                               !list member structure
@lm := _NULL;
WHILE _NOTNULL (@lm := _SUCCESSOR^LM (list,lm)) DO
  BEGIN
  ...
  END;
```

_NULL^LIST

_NULL ^LIST initializes a list structure. It is useful to initialize a _LIST declared in an uninitialized memory area.

```
CALL _NULL^LIST( list );
```

list input/output

is the name of a _LIST.

Considerations

_NULL^LIST does not deallocate an existing list. Use _DEALLOCATE^LIST to remove and deallocate all existing list members.

list must not have any members prior to the operation. If it does, these members will not be accessible later on, because the pointer to them will be initialized.

Example

```
_LIST (worklist);
CALL _NULL^LIST(worklist);
```

OBJECTLIST

OBJECTLIST is the name assigned to the input and output object lists by the _COMMAND^CONTEXT^HEADER define.

Example

The following example gets a member off the input list and creates a member on the output list:

```
STRUCT input^lm^def (*);
  BEGIN
    _INPUT^LM^HEADER;
   ...
  END;

STRUCT output^lm^def (*);
  BEGIN
    _OUTPUT^LM^HEADER;
   ...
  END;

STRUCT command^context^def (*);
  BEGIN
    _COMMAND^CONTEXT^HEADER;
    INT .EXT inobj (input^lm^def);
    INT .EXT outobj (output^lm^def);
  END;

! Thread proc locals!

INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT in (input^def) := @cx._INPUT;
INT .EXT out (output^def) := @cx._OUTPUT;
       .
       .
! Get the next input object

IF _ISNULL( @cx.inobj := _GET^LM (in.OBJECTLIST))
  THEN RETURN _RC^STOP;   ! out of input list members

! Create output list member

IF _ISNULL (@cx.outobj := _PUT^LM (out.OBJECTLIST,,
                                   $LEN (cx.outobj)))
  THEN ... < out of available memory > ;
       .
       .
```

_OFF

_OFF is a Boolean define statement that is TRUE if any one-bit of *bit-mask* is off in *int-exp*. TRUE is nonzero, not necessarily -1.

The _OFF function is the same as the _ALLOFF function. It is more descriptive to use _ALLOFF when testing more than one bit.

```
_OFF ( int-exp , bit-mask )
```

int-exp input

INT:value

is the INT expression compared with *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-exp* to test.

Example

The following example tests if the _EV^STARTUP bit is off in _LAST^EVENTS:

```
IF _OFF (_LAST^EVENTS, _EV^STARTUP)
  THEN ... ;
```

_ON

_ON is a Boolean define statement that is TRUE if any one-bit of *bit-mask* is on in *int-exp*. TRUE is nonzero, not necessarily -1.

The _ON function is the same as the _ANYON function. It is more descriptive to use _ANYON when testing more than one bit.

```
_ON ( int-exp , bit-mask )
```

int-exp input

INT:value

is the variable compared with *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-exp* to test.

Example

The following example tests if the _EV^CANCEL bit is on in _LAST^EVENTS:

```
IF _ON (_LAST^EVENTS, _EV^CANCEL)
  THEN ...;
```

_OPEN^CI

_OPEN^CI opens a CI for communication.

```
error := _OPEN^CI ( ci-config
                   ,ciid
                   ,[ processname ]
                   ,[ nowait-depth ])
```

error returned value

INT

If > 0, is the file system error returned from the FILE_OPEN_ procedure. File system errors are described in the *Guardian Procedure Errors and Messages Manual*.

If < 0, is a ZDSN error. See Appendix B, “DSNM Error Codes,” for ZDSN error code definitions.

ci-config input

is the extended pointer (declared in globals). It points to the _CI^DEF-defined CI configuration structure, containing CI configuration parameters.

ciid input

is the CIID structure (declared with _CI^ID), identifying an open CI.

processname input

STRING .EXT

is the name of an existing CI process. The process name is in external format and must be terminated with a null or a blank.

nowait-depth input

INT:value

is the maximum number of concurrent _SEND^CI operations that can be executed against this CI by this thread. The default is 1, if *nowait-depth* is omitted or specified as a value < 1.

Considerations

- Your _STARTUP procedure must call _ADD^CI to fill the _CI^DEF-defined CI configuration structure for each CI class opened by the command thread.
- *ci-config* and *ciid* play a role in command-thread CI communication that is analogous to file name and file number in Tandem NonStop Kernel interprocess communications. A CI is identified by its *ci-config*; a *ciid* refers to a particular instance of an open CI.

- To communicate with a server CI, you must allocate a message buffer large enough to hold the larger of the message and its response. This buffer must be in the command context space or in an allocated list member. It cannot be in globals or procedure locals. If more than one operation is to be outstanding (whether on the same or on separate CIs), you should also supply an INT(32) tag for the operation, usually a pointer to some identifying data.
- After initiating a request for CI communication, the thread must return to the frame to wait for its completion with a RETURN _RC^WAIT. When the communication is complete, the frame dispatches the thread with the event _EV^IODONE.
- If multiple _SEND^CIs are outstanding concurrently, they are completed one at a time and dispatched with _EV^IODONE. Threads must return to the frame with _RC^WAIT to obtain completions of subsequent operations.

Example

The following example opens a CI:

```
< within user globals area >

STRUCT context^def (*);
  BEGIN                  ! Command thread context definition
    _COMMAND^CONTEXT^HEADER;
    INT .EXT input^lm (input^lm^def);   ! Current input list
                                        ! member
    INT .EXT output^lm (out^lm^def);    ! Current output list
                                        ! member
    _CI^ID (current^ci);
    INT cibuf[0:7];
  END;

    INT .EXT scp (_CI^DEF);
    INT .EXT snaxcdf (_SUBSYS^DEF);

    STRING .scpclass[0:ZDSN^MAX^CICLASS-1] := [“SCP ”];
    STRING .cdf[0:ZDSN^MAX^SUBSYS-1] := [“SNAXCDF ”];

< within _STARTUP procedure >
      .
    IF _ISNULL (@scp := _ADD^CI (scpclass)) THEN
      RETURN ZDSN^ERR^INTERNAL^ERR;
    IF _ISNULL (@snaxcdf := _ADD^SUBSYS (cdf)) THEN
      RETURN ZDSN^ERR^INTERNAL^ERR;
     .
     .
< within command thread >
    .
```

```
INT .EXT cx (context^def) = _THREAD^CONTEXT^ADDRESS;
INT error, cmd^len;
LITERAL max^cmd = ..., max^reply = ...;
  IF (error := _OPEN^CI (scp, cx.current^ci,
                         cx.input^lm.FOBJ.Z^MANAGER);
    THEN ... < open error > ;
    .
    .
```

_OUTPUT

_OUTPUT is the name assigned to the _OUTPUT^DEF structure template within the command context area where the frame declares the output object list (OBJECTLIST). _OUTPUT is assigned by _COMMAND^CONTEXT^HEADER.

Example

The following example of a local data definition gives a thread procedure access to the output area:

```
INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT out (_OUTPUT^DEF) := @cx._OUTPUT;
```

See the _FOBJECT^INIT description for another _OUTPUT example.

_OUTPUT^DEF

_OUTPUT^DEF is a structure template where the frame declares the output object list (OBJECTLIST).

```
STRUCT _OUTPUT^DEF (*);
  BEGIN
  _LIST (OBJECTLIST);
  END;
```

Example

The following example of a local data definition gives a thread procedure access to the output area:

```
INT .EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT out (_OUTPUT^DEF) := @cx._OUTPUT;
```

See the _FOBJECT^INIT description for another _OUTPUT^DEF example.

_OUTPUT^LM^HEADER

_OUTPUT^LM^HEADER describes the first part of the user-defined output list member structure. It is required as part of the output list member definition.

_OUTPUT^LM^HEADER generates a formatted object structure (ZDSN^DDL^FOBJECT^DEF); it identifies this object structure as FOBJECT and identifies other fields for the frame.

```
_OUTPUT^LM^HEADER;
```

The following FOBJECT fields should be completed by the command thread before releasing an output object list member to the frame:

Z^RESULT Is the result code for this object.

Z^SUBSYS Is the name of the subsystem.

Z^OBJTYPE Is the object type.

Z^OBJNAME Is the object name, blank-filled.

Z^MANAGER Is the name of the manager, if any, blank-filled.

It is not necessary to fill in ZOBJNAME^OCCURS nor Z^MANAGER^OCCURS.

See Section 4, “DSNM Command Requirements,” for more information on FOBJECT fields.

Example

Following is an example of an output list member structure declaration:

```
STRUCT output^list^member^def (*);
  BEGIN
    _OUTPUT^LM^HEADER;
    user-defined-area
   ...
  END;
```

See the _FOBJECT^INIT description for another _OUTPUT^LM^HEADER example.

_POP^LM

_POP^LM removes the current last member (most recently added) from a list and returns its address. If a list is empty, _POP^LM returns _NULL.

```
@list-member := _POP^LM ( list
                        ,[ length ] );
```

list-member returned value

INT .EXT

is the address of the removed member.

list input

is the name of a _LIST.

length output

INT:ref

returns the length of the removed member, in bytes.

Considerations

- Removing a member with _POP^LM does not immediately deallocate memory. The removed member's memory remains allocated and its contents useable until the next successive member is removed with _POP^LM, or a new member is added with _PUSH^LM.
- The removed member does not participate in list scans with _SUCCESSOR^LM nor _PREDECESSOR^LM.
- _PUT^LM, if used with _POP^LM, also deallocates memory for the last element removed by _POP^LM.
- Normally, a list is processed either by _PUT^LM plus _GET^LM or by _PUSH^LM plus _POP^LM, but not both.
- _UNPOP^LM replaces the last list member removed by _POP^LM.

Example

In the following example, the latest member of `outlist` is removed and `list^member` is set to point to it:

```
_LIST (outlist);
INT .EXT list^member (list^member^def);
INT length;
IF _ISNULL (@list^member := _POP^LM (outlist, length));
  THEN ... < list empty > ;
```

_POP^THREAD^PROCSTATE

_POP^THREAD^PROCSTATE restores the values of the thread procedure and thread state saved with the most recent _PUSH^THREAD^PROCSTATE.

```
error := _POP^THREAD^PROCSTATE ;
```

error returned value

INT

is a ZDSN^ERR value indicating the outcome of the call. See Appendix B, “DSNM Error Codes,” for error code definitions.

Example

See the example for _PUSH^THREAD^PROCSTATE.

_PREDECESSOR^LM

_PREDECESSOR^LM returns the address of the list member placed on the list immediately before the current list member was added.

```
@prev-list-member := _PREDECESSOR^LM ( list
                                      ,list-member );
```

prev-list-member returned value

INT .EXT

is the address of the predecessor of the current list member.

list input

is the name of a _LIST.

list-member input

INT .EXT

is a pointer to a current member of *list*.

Considerations

- List members are logically ordered. The first (or front or head) member is the earliest put on the list and the last (or end or tail) member is the latest. Each member has a successor and a predecessor, the predecessor of the first and the successor of the last being _NULL.
- *list-member* must be a current member of *list*, or @*list-member* must be _NULL. If @*list-member* is _NULL, the address of the last member of *list* is returned.
- Predecessor list members are not necessarily stored at decreasing memory addresses. You cannot determine the order of list members by comparing their addresses.
- _PREDECESSOR^LM returns _NULL if one of the following is true:
  - *list-member* is the first member of *list*.
  - *list* is empty.
  - An error occurs.

Example

The following example sets two pointers, one to the last member of `list`, and another to the next-to-last member:

```
_LIST (list);
INT .EXT lm (list^member^def);     !extended pointer to list
                                   !member structure
INT .EXT prevlm (list^member^def); !another extended pointer
                                   !to list member struct
@lm := _LAST^LM (list);
@prevlm := _PREDECESSOR^LM (list, lm);
```

_PRIVATE^THREAD^EVENT

_PRIVATE^THREAD^EVENT produces an INT constant with a single one-bit, suitable for labeling an event to look different from any frame-generated event.

```
_PRIVATE^THREAD^EVENT ( num );
```

num is a number in the range 0 through 7.

Considerations

- Currently, the thread can declare eight events guaranteed to be different from all frame-generated events.
- Thread procedures must call _SIGNAL^EVENT to generate private events. When the thread generates its own event(s), it is redispatched immediately when it returns _RC^WAIT to the frame.

Example

The following example causes two user-defined events, `sub^object` and `next^object`, to be turned on in _LAST^EVENTS at the next dispatch:

```
LITERAL next^object = _PRIVATE^THREAD^EVENT (0);
LITERAL sub^object = _PRIVATE^THREAD^EVENT (1);

CALL _SIGNAL^EVENT (sub^object + next^object);
RETURN _RC^WAIT;
   .
!After the next dispatch ...
   .
IF _ALLON (_LAST^EVENTS, sub^object + next^object)
   THEN ...;
```

_PROCESS^PARAMS

_PROCESS^PARAMS is a global structure defined in the I process globals in which the frame stores standard process parameters it retrieves as part of its startup function.

```
STRUCT _PROCESS^PARAMS (ZDSN^DDL^PROCESS^PARAMS^DEF);
```

Considerations

The contents of the _PROCESS^PARAMS structure is as follows:

```
DEFINITION ZDSN-DDL-PROCESS-PARAMS.
  02 Z-CLASS-OCCURS      TYPE ZSPI-DDL-UINT.
  02 Z-CLASS             TYPE ZDSN-DDL-CLASS.
  02 Z-COMPONENT-OCCURS  TYPE ZSPI-DDL-UINT.
  02 Z-COMPONENT         TYPE ZDSN-DDL-COMPONENT.
  02 Z-MYSYSTEM-OCCURS   TYPE ZSPI-DDL-UINT.
  02 Z-MYSYSTEM          TYPE ZDSN-DDL-SYSTEM.
  02 Z-MYREALSYSTEM-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-MYREALSYSTEM      TYPE ZDSN-DDL-SYSTEM.
  02 Z-MYPROCESS-OCCURS  TYPE ZSPI-DDL-UINT.
  02 Z-MYPROCESS         TYPE ZDSN-DDL-PNAME.
  02 Z-TESTMODE          TYPE ZSPI-DDL-INT.
  02 Z-DEBUG-LEVEL       TYPE ZSPI-DDL-ENUM.
  02 Z-SECTION-NAME-OCCURS TYPE ZSPI-DDL-UINT.
  02 Z-SECTION-NAME      TYPE ZDSN-DDL-PARAMNAME.
END
```

_PUSH^LM

_PUSH^LM allocates memory for a new last member of a list and returns its address. _NULL is returned if no memory is available for a new list member.

```
@list-member := _PUSH^LM ( list
                         ,[ length ]
                         ,initlength
                         ,[ initdata ] );
```

list-member returned value

INT .EXT

is the address of the new member.

list input

is the name of a _LIST.

length input

INT:value

is the length of the new member, in bytes.

initlength input

INT:value

is the number of bytes of the new member to be initialized (to the contents of *initdata*, if present); otherwise, it is initialized to 0s.

initdata input

INT .EXT

is a structure or an array containing initial data for the list member.

Considerations

- If *length* is not provided, *initlength* is taken as the length of the new member, as well as the initializing length.
- Normally, a list is processed either by _PUT^LM plus _GET^LM or by _PUSH^LM plus _POP^LM, but not both.
- _PUSH^LM deallocates and reuses the memory assigned to the last element removed by _POP^LM.

Example

The following example allocates space for a new list member initialized to binary 0s:

```
_LIST (list);
INT .EXT lm (list^member^def);

IF _ISNULL (@lm := _PUSH^LM (list,,$LEN (lm)))
  THEN <no memory available> ;
```

_PUSH^THREAD^PROCSTATE

_PUSH^THREAD^PROCSTATE saves the current thread procedure and thread state, and optionally sets new values for the current thread procedure and thread state.

```
error := _PUSH^THREAD^PROCSTATE ( [ @procname ]
                                 ,[ state ] );
```

error returned value

INT

is a ZDSN^ERR value, indicating the outcome of the call. See Appendix B, "DSNM Error Codes," for error code definitions.

procname input

is a thread procedure that becomes the current thread procedure after the existing thread procedure is saved.

state input

INT:value

is an INT expression that becomes the current thread state after the existing value is saved.

Considerations

- If *procname* and *state* are not provided, the current thread procedure and thread state are saved, and no new current thread procedure and state are set.
- The current thread procedure is defined as the procedure called by the frame the next time the thread is dispatched.
- _POP^THREAD^PROCSTATE restores the values of the thread procedure and thread state saved with the last _PUSH^THREAD^PROCSTATE.

Example

In the following example, the frame dispatches PROC^X of the command thread in _ST^INITIAL. PROC^X calls PROC^Y in STATE^B by:

- Setting its return state to STATE^A.
- Saving the old current thread procedure and state values, and setting new current thread procedure and thread state values.
- Signaling an event and returning to the frame to dispatch the new thread procedure PROC^Y in the new state STATE^B.

PROC^Y checks for event EV^STARTUP, resets the current thread procedure and thread state to the previously saved values of PROC^X and STATE^A, and returns to the frame to dispatch PROC^X in STATE^A.

```
_THREAD^PROC (PROC^X);
  BEGIN
  .
  .
  CASE _THREAD^STATE OF
    BEGIN
      _ST^INITIAL -->
        _THREAD^STATE := STATE^A;
        IF (error := _PUSH^THREAD^PROCSTATE(@PROC^Y,STATE^B))
          THEN ... < error > ;
        CALL _SIGNAL^EVENT (_EV^STARTUP);
        RETURN _RC^WAIT;

      STATE^A -->
          .
          .
        RETURN _RC^STOP;
    END;
  _END^THREAD^PROC;

_THREAD^PROC (PROC^Y);
  BEGIN
  .
  .
  CASE _THREAD^STATE OF
    BEGIN
      STATE^B -->
        IF _ON (_LAST^EVENTS, _EV^STARTUP)
          THEN
            BEGIN
               .
               .
              IF (error := _POP^THREAD^PROCSTATE)
                THEN ... < error > ;
              CALL _SIGNAL^EVENT (_EV^CONTINUE);
              RETURN _RC^WAIT;
            END;
    END;
  _END^THREAD^PROC;
```

_PUT^LM

_PUT^LM allocates memory for a new last member of a list and returns its address. _NULL is returned if no memory is available for a new list member.

```
@list-member := _PUT^LM ( list
                        ,[ length ]
                        ,initlength
                        ,[ initdata ] );
```

list-member returned value

INT .EXT

is the address of the new member.

list input

is the name of a _LIST.

length input

INT:value

is the length of the new member, in bytes.

initlength input

INT:value

is the number of bytes of the new member to be initialized (to the contents of *initdata*, if present): otherwise, it is initialized to 0s.

initdata input

INT .EXT

is a structure or an array containing initial data for the list member.

Considerations

- If *length* is not provided, *initlength* is taken as the length of the new member, as well as the initializing length.
- Normally, a list is processed either by _PUT^LM plus _GET^LM or by _PUSH^LM plus _POP^LM, but not both.
- _PUT^LM, if used with _POP^LM, deallocates and reuses the memory assigned to the last element removed by _POP^LM.

Example

In the following example, the INIT^LM^VALUES structure is set to initializing values for each worklist member; then a list member is allocated and initialized to INIT^LM^VALUES:

```
STRUCT .init^lm^values (lm^def);
_LIST (worklist);
INT .EXT lm (lm^def);

IF _ISNULL(@lm :=_PUT^LM (worklist,,$LEN(lm),init^lm^values))
  THEN ... <memory error> ;
```

See the _FOBJECT^INIT description for another _PUT^LM example.

_RC^ABORT

The thread returns _RC^ABORT to the frame when a command abnormally terminates.

```
RETURN _RC^ABORT ( error );
```

error

INT

is a ZDSN^ERR value, indicating the reason for the abnormal command termination. See Appendix B, “DSNM Error Codes,” for error code definitions.

_RC^NULL

_RC^NULL is a special return code that is not equal to any valid thread return code; it must be returned to the frame in _COMMAND^TERMINATION^PROC.

_RC^NULL may be returned by an _RC^TYPE procedure (defined later in this appendix) to indicate that it has not returned any valid frame return code.

_RC^NULL must not be returned to the frame by any thread procedure.

```
_RC^NULL;
```

_RC^STOP

The thread returns _RC^STOP to the frame when a command ends normally.

```
RETURN _RC^STOP;
```

A command that terminates early due to an _EV^CANCEL event from the frame is considered to have terminated normally; the thread should return _RC^STOP to the frame.

_RC^TYPE

_RC^TYPE declares function procedures that can be called by a thread procedure (but are not themselves thread procedures) and that return a frame return code value.

_RC^TYPE also declares variables to hold the frame return code (_RC^) values such as values returned by _RC^TYPE function procedures.

```
_RC^TYPE PROC procname ;

_RC^TYPE var1, [ var2 [,...]];
```

The special return code _RC^NULL may be returned by an _RC^TYPE procedure to indicate that it has not returned any valid frame return code. _RC^NULL must not be returned to the frame by any thread procedure.

Example

In this example, a thread procedure calls an _RC^TYPE procedure. The called procedure returns a frame return code, which is interpreted by the calling procedure.

```
_RC^TYPE PROC process^object ( ... );
  BEGIN
    .
    .
  END;

_THREAD^PROC (_COMMAND^PROC);
  BEGIN
    _RC^TYPE obj^rc;
      .
      .
      obj^rc := process^object ( ... );

      IF obj^rc <> _RC^NULL
        THEN
          RETURN obj^rc;
      .
      .
  _END^THREAD^PROC;
```

_RC^WAIT

The thread returns _RC^WAIT to the frame to wait for the next event.

```
RETURN _RC^WAIT;
```

_REAL^LAST^EVENTS

_REAL^LAST^EVENTS is set each time the command thread is dispatched to contain the event(s) that caused the current dispatch. Each bit represents a different event.

```
_REAL^LAST^EVENTS
```

_REAL^LAST^EVENTS is a define that returns a value and as such, can only be tested; it cannot be altered. _LAST^EVENTS, which is also set to the current event(s) at each dispatch, is a global variable that can be tested and altered.

When a thread is invoked for the first time, _LAST^EVENTS and _REAL^LAST^EVENTS are set to _EV^STARTUP.

Example

The following example tests if the _EV^IODONE bit is on in _REAL^LAST^EVENTS:

```
IF _ON (_REAL^LAST^EVENTS, _EV^IODONE)
  THEN ...;
```

_RELEASE^OUTPUT

_RELEASE^OUTPUT releases a member of the output list to the frame. Once released, the output list member can be removed by the frame at the next frame return.

Each output list member should be released as soon as it is completely completed.

```
_RELEASE^OUTPUT ( output-list-member );
```

output-list-member input

INT .EXT

is the output list member released to the frame.

Considerations

- The frame cannot remove an output list member that has an unreleased predecessor.
- Thread termination releases all output list members.

Example

See the _FOBJECT^INIT description for a _RELEASE^OUTPUT example.

_REPORT^INTERNAL^ERROR

_REPORT^INTERNAL^ERROR logs internal errors to the $0 EMS collector.

```
_REPORT^INTERNAL^ERROR ([ internalcode ]
                       ,[ severity ]
                       ,[ I1 ]
                       ,[ I2 ]
                       ,[ I3 ]
                       ,[ I4 ] );
```

internalcode input

INT:value

is an internal code that you assign strictly for your own use.

severity input

INT:value

is one of the following values, indicating the severity of the event logged to EMS:

_EMS^EVENT^INFO	The event is a nonfatal informative message; it is defined earlier in this appendix.
_EMS^EVENT^CRITICAL	The event reports a critical but nonfatal condition; it is defined earlier in this appendix.
_EMS^EVENT^FATAL	The event reports a fatal condition; it is defined earlier in this appendix.

I1 .. I4 input

INT:value

are four optional integer values for you to report relevant information.

Considerations

- The internal code is designed to assign a code for locating the point of the error in the program.
- The integer values are designed to display internal data values in the event message to help trace the error.

Example

```
IF _NOTNULL (@inobj := _GET^LM (cx.current^in)) THEN
  BEGIN
    CALL _REPORT^INTERNAL^ERROR (1, _EMS^EVENT^INFO);
    RETURN _RC^ABORT (ZDSN^ERR^INTERNAL^ERR);
  END;
```

_REPORT^STARTUP^ERROR

_REPORT^STARTUP^ERROR reports fatal startup errors to the $0 EMS collector, resulting from _STARTUP procedure startup parameter or configuration errors.

```
_REPORT^STARTUP^ERROR ([ internalcode ]
                      ,[ severity ]
                      ,[ text ] );
```

internalcode input

INT:value

is an internal code that you can assign strictly for your own use.

severity input

INT:value

is one of the following values, indicating the severity of the event logged to EMS:

_EMS^EVENT^INFO	The event is a nonfatal informative message; it is defined earlier in this appendix.
_EMS^EVENT^CRITICAL	The event reports a critical but nonfatal condition; it is defined earlier in this appendix
_EMS^EVENT^FATAL	The event reports a critical and fatal condition; it is defined earlier in this appendix.

text input

STRING .EXT

is the error text. The text string must be terminated by a null character.

Considerations

The internal code is designed to assign a code for locating the point of the error in the program.

Example

```
INT PROC _STARTUP (cxl, inputl) EXTENSIBLE;
INT .cxl, .inputl;
  BEGIN
   STRING errtext[0:29] :=["Invalid SPIFFY configuration",0];
   cxl := $LEN (cx^def);    ! Command thread context length
   inputl := $LEN (object^lm^def);   ! Frame input object
                                     ! list member length

     ! Get CI and subsystem configurations
   IF _ISNULL (@spifmon := _ADD^CI (spifclass))
      OR _ISNULL (@spiffy := _ADD^SUBSYS (spifsys))
   THEN CALL _REPORT^STARTUP^ERROR (0, _EMS^EVENT^FATAL,
                                    errtext);
   RETURN ZDSN^ERR^NOERR;
  END;
```

_RESTORE^THREAD^AND^DISPATCH

_RESTORE^THREAD^AND^DISPATCH restores the thread procedure and state last pushed and returns to the frame for immediate dispatch with the specified event.

```
_RESTORE^THREAD^AND^DISPATCH ( [ event ] );
```

event input

INT:value

is an INT expression that designates the event(s) with which the restored procedure is to be dispatched. The default is _EV^CONTINUE.

When using _RESTORE^THREAD^AND^DISPATCH with all arguments omitted (accepting the default event), you must use the following construction:

```
_RESTORE^THREAD^AND^DISPATCH ( );
```

_RESTORE^THREAD^AND^DISPATCH can fail only with error ZDSN^ERR^NOTPUSHED. When a failure occurs, code immediately following the function is executed.

Example

```
IF _ISNULL(@inobj := @cx.currentobj :=
   _GET^LM(cx.current^in))
  THEN
    BEGIN
      ! Out of input objects; restore caller and continue.
      ! Note:  Calling proc has set the state in which it
      !        desires to return before saving the thread
      !        state and dispatching this proc.
      _RESTORE^THREAD^AND^DISPATCH (_EV^CONTINUE);
         ! If _RESTORE^THREAD^AND^DISPATCH fails,
         ! we fall through to here and ...
      RETURN _RC^ABORT (ZDSN^ERR^NOTPUSHED);
    END;
```

_SAVE^THREAD^AND^DISPATCH

_SAVE^THREAD^AND^DISPATCH saves the current thread procedure and state, optionally sets new current thread procedure and state values, and returns to the frame for immediate dispatch.

```
_SAVE^THREAD^AND^DISPATCH ( [ @procname ]
                          ,[ state ]
                          ,[ event ] );
```

procname input

is the dispatched thread procedure. This procedure becomes the new current thread procedure. The default is to redispatch the existing current thread procedure.

state input

INT: value

is an INT expression that designates what becomes the current thread state when the specified thread is dispatched. The default is to keep the existing current state.

event input

INT:value

is an INT expression that designates the event(s) with which the new procedure is dispatched. The default is _EV^CONTINUE.

When using _SAVE^THREAD^AND^DISPATCH with all arguments omitted (accepting all the defaults), you must use the following construction:

```
_SAVE^THREAD^AND^DISPATCH ( );
```

_SAVE^THREAD^AND^DISPATCH can fail only with the error ZDSN^ERR^MEMORY. When a failure occurs, code immediately following the function is executed.

Example

```
IF _ON (inobj.cf, c^info) THEN
  BEGIN
        ! Set state where we wish to return to this proc.
    _THREAD^STATE := st^done;
    _SAVE^THREAD^AND^DISPATCH (@info^proc, st^new^object,
                               _EV^STARTUP);
        ! If _SAVE^THREAD^AND^DISPATCH fails,
        ! we fall through to here and ...
    RETURN _RC^ABORT (ZDSN^ERR^MEMORY);
END;
```

_SEND^CI

_SEND^CI initiates sending a message to a server CI. After initiating _SEND^CI, the thread must eventually return to the frame to wait for its completion with _RC^WAIT.

```
error := _SEND^CI ( ciid
                   ,buffer
                   ,write-count
                   ,reply-count
                   ,[ context-boolean ]
                   ,[ tag ]
                   ,[ timeout ] );
```

error returned value

INT

If > 0, is a file system error. File system errors are described in the *Guardian Procedure Errors and Messages Manual*.

If < 0, is a ZDSN error. See Appendix B, “DSNM Error Codes,” for ZDSN error code definitions.

ciid input

is the CIID structure (declared with _CI^ID) identifying an open CI.

buffer input

INT .EXT:ref:*

is an array containing information to be sent to the CI. On return, *buffer* contains the information read from the CI (and is referred to as the “reply buffer”).

write-count input

INT:value

is the number of bytes to send to the CI.

reply-count input

INT:value

is the maximum number of bytes accepted from the CI in the reply buffer.

context-boolean input

INT:value

indicates whether the send is context-free, meaning it does not depend on any previous communication with this *ciid*. Specifically:

- If *context-boolean* is 0 (FALSE), the send is context-free, which means it may be sent to a new instance of the same CI if an error occurs.
- If *context-boolean* is nonzero (TRUE) or omitted, the command is assumed to be contextually dependent on earlier commands sent to this CI, and the frame will not send the command to a new instance of the CI if an error occurs.

tag input

INT(32):value

uniquely identifies this _SEND^CI operation. *tag* is used to distinguish among sends if more than one operation is outstanding at the same time (whether to the same or to separate CIs). Normally, a tag is the address of a list member in which the user places identifying information about the operation.

timeout input

INT(32):value

is the time in .01-second units that this I/O operation is allowed to remain outstanding without a response. If the CI does not respond within this time, the I/O operation completes with file system error 40 (cannot be retried).

If *timeout* is omitted or less than or equal to 0D, an indefinite wait is indicated.

Considerations

- If the communication is an SPI message containing a context token from a previous communication, the message is contextually dependent on the previous message, even though the CI is context-free. In this case, *context-boolean* must be true.
- The frame dispatches the thread with an _EV^IODONE event when a _SEND^CI operation completes. At that time:

_CI^LASTERROR (*ciid*)	Is the INT file system error of the operation.
_CI^REPLYLENGTH (*ciid*)	Is the INT length of the reply.
_CI^REPLYADDRESS (*ciid*)	Is the INT(32) extended address of the reply.
_CI^REPLYTAG (*ciid*)	Is the INT(32) tag of the operation.
_CI^FILENUM (*ciid*)	Is the INT Guardian file number of the CI.

Example

In the following example, the command thread initiates a _SEND^CI request and returns to the frame to wait for an I/O completion event:

```
< within user globals area >

STRUCT context^def (*);
  BEGIN                  ! Command thread context definition
    _COMMAND^CONTEXT^HEADER;
    INT .EXT input^lm (input^lm^def);   ! Current input list
                                        ! member
    INT .EXT output^lm (out^lm^def);  ! Current output list
                                        ! member
    _LIST (worklist);
    _CI^ID (current^ci);
    INT cibuf[0:<buffer word length>)];
  END;

< within command thread >

INT .EXT cx (context^def) = _THREAD^CONTEXT^ADDRESS;
INT .EXT currentobj (input^lm^def);
INT error, cmd^len;
LITERAL max^cmd = ..., max^reply = ...;

  IF _ISNULL (@currentobj := _FIRST^LM(cx._INPUT.OBJECTLIST))
    THEN ... < empty list > ;

  IF (error := _OPEN^CI (ci^config, cx.current^ci,
                         cx.input^lm.FOBJ.Z^MANAGER);
    THEN ... < open error > ;
        .
< Allocate buffer for _SEND^CI >

  IF _ISNULL (@cx.cibuf := _PUT^LM (cx.worklist,,
                                    $MAX(max^cmd,max^reply)))
    THEN ... < memory error > ;
            .
< Construct buffer to execute command when sent to CI >
              .
  IF (error := _SEND^CI (cx.current^ci, cx.cibuf, cmd^len,
                         max^reply,0));
    THEN ... < send error > ;

  RETURN _RC^WAIT;                    ! Wait for EV^IODONE
```

_SET^THREAD^PROC

_SET^THREAD^PROC sets the current thread procedure to be called by the frame at the next thread dispatch.

```
_SET^THREAD^PROC ( @procname );
```

procname input

is the name of the thread procedure called by the frame at the next thread dispatch.

Considerations

- Setting the current thread procedure is a high-level state change. For instance, the initial thread procedure (_COMMAND^PROC) might examine the command passed to it by the frame (when first dispatched) to determine if it is an informational command or a state-change command. Since these commands have considerably different output requirements, it may be convenient to have different procedures perform their processing (see the provided example).
- The current thread procedure and thread state may be saved and later restored with combinations of _DISPATCH^THREAD, _PUSH^THREAD^PROCSTATE, _POP^THREAD^PROCSTATE, _SAVE^THREAD^AND^DISPATCH, and _RESTORE^THREAD^AND^DISPATCH.

Example

The following example causes the thread procedure selected for the command type to be dispatched immediately with the event _EV^STARTUP:

```
_THREAD^PROC (info^thread^proc);
  BEGIN
  < procedure body >
  _END^THREAD^PROC;

_THREAD^PROC (state^change^thread^proc);
  BEGIN
  < procedure body >
  _END^THREAD^PROC;

_THREAD^PROC (_COMMAND^PROC);
  BEGIN
    .
    IF info-type-command
      THEN _SET^THREAD^PROC (@info^thread^proc)
      ELSE _SET^THREAD^PROC (@state^change^thread^proc);
    CALL _SIGNAL^EVENT (_EV^STARTUP);
    RETURN _RC^WAIT;
    .
  _END^THREAD^PROC;
```

_SET^TIMEOUT

_SET^TIMEOUT allows the command thread to delay for a time interval by arranging for a future timeout event.

```
CALL _SET^TIMEOUT ( time-interval
                   ,[ tag ] );
```

time-interval input

INT(32):value

specifies the timeout period, in .01-second units. This value must be greater than 0.

tag input

INT(32):value

is an identifier associated with the timer, which is placed into command context.

After the interval is set, the thread must return to the frame with _RC^WAIT. The thread is dispatched with _EV^TIMEOUT when the interval elapses. If supplied, the tag is placed into command context and can be accessed with _LAST^TIMEOUT^TAG. Usually, a timeout tag is the address of a list member holding information about the purpose of the timeout.

Example

The following example dispatches the current thread procedure with _EV^TIMEOUT after a 1.00 second delay:

```
LITERAL asec = 100D;

CALL _SET^TIMEOUT (asec);          !Wait one second
RETURN _RC^WAIT;                   !Wait for _EV^TIMEOUT
```

See the _LAST^TIMEOUT^TAG description for another _SET^TIMEOUT example.

_SIGNAL^EVENT

_SIGNAL^EVENT generates private events or simulates frame events.

```
CALL _SIGNAL^EVENT ( event(s) );
```

event(s) input

INT:value

is an INT expression, the one-bits of which designate the event(s) to be generated.

Considerations

- When the thread generates its own event(s) with _SIGNAL^EVENT, it is redispatched immediately when it returns _RC^WAIT to the frame.
- You may simulate any frame event by signaling it with _SIGNAL^EVENT. For example:

  ```
  CALL _SIGNAL^EVENT (_EV^IODONE);
  ```

  When you simulate a frame event, be careful not to use control variables set by frame-generated events (such as _LAST^CI^ID or _LAST^TIMEOUT^TAG), unless they are set to match the event simulated.
- Events generated by the frame occur singly, with one dispatch per event. All events generated by the thread occur together, immediately after the next return to the frame and before any frame-generated events.

Example

The following example causes two user-defined events, `sub^object` and `next^object`, to be turned on in _LAST^EVENTS at the next dispatch:

```
LITERAL next^object = _PRIVATE^THREAD^EVENT (0);
LITERAL sub^object = _PRIVATE^THREAD^EVENT (1);

CALL _SIGNAL^EVENT (sub^object + next^object);
RETURN _RC^WAIT;
   .
!After the next dispatch ...
   .
IF _ALLON (_LAST^EVENTS, sub^object + next^object)
   THEN ...;
```

Since _PRIVATE^THREAD^EVENT generates a bit value different from all frame events, the events that are on in _LAST^EVENTS mean this dispatch was caused by _SIGNAL^EVENT.

_STARTUP

_STARTUP is a user-provided initialization procedure called by the frame. It supplies the lengths of the user context area and input list members, and retrieves and places subsystem and CI configuration parameters into predefined structures for use by the frame.

You must call _ADD^SUBSYS in your _STARTUP procedure for each subsystem your I process handles, as well as _ADD^CI for each CI class with which your I process communicates.

```
INT PROC _STARTUP ( context-length , input-lm-length )
  EXTENSIBLE;
```

context-length output

INT:ref

is the length of the command context structure, in bytes.

input-lm-length output

INT:ref

is the length of an input list member, in bytes.

The frame must know the lengths of the user-defined command context and input list member structures, since it allocates these areas before it creates the first instance of the command thread.

If no values are provided for *context-length* and *input-lm-length*, the frame allocates only the space required for its own use (as defined by __COMMAND^CONTEXT^HEADER and _INPUT^LM^HEADER). No space is reserved for user data.

Example

The following example of an initialization procedure assumes that the user has defined structure templates for the command context area (command^context^def), an input list member (input^lm^def), pointers to a _CI^DEF-defined CI configuration structure (scp), and a _SUBSYS^DEF-defined subsystem configuration structure (snaxcdf):

```
INT PROC _STARTUP (cx^length, in^lm^length) EXTENSIBLE;
INT .cx^length, .in^lm^length;

 BEGIN
  cx^length := $LEN (command^context^def);
  in^lm^length := $LEN (input^lm^def);
```

```
    IF _ISNULL (@scp := _ADD^CI (scpclass)) THEN
      RETURN ZDSN^ERR^INTERNAL^ERR;
    IF _ISNULL (@snaxcdf := _ADD^SUBSYS (cdf)) THEN
      RETURN ZDSN^ERR^INTERNAL^ERR;
 RETURN ZDSN^ERR^NOERR;
END;
```

_STARTUP^MODE

_STARTUP^MODE is a user-provided procedure called by the frame when it begins startup processing. _STARTUP^MODE retrieves the component name of the subsystem(s) being handled by the I process and determines if the I process is running in test mode and whether to use the COMPONENT process parameter value (if one appears in the startup message).

```
INT PROC _STARTUP^MODE ( component
                       , testmode
                       , accept-startup-component
                       , subject )
                        EXTENSIBLE;
```

component output

STRING .EXT ! (ZDSN^DDL^COMPONENT^DEF) !

is the name, left justified, blank-filled, of the subsystem handled by the I process.

component is used for configuration parameter retrieval, and is usually the name of the subsystem that the I process handles. For I processes that handle multiple subsystems, component is an arbitrary name chosen by the developer of the process. For example, the Tandem-supplied SCP I process handles multiple communications subsystems: COMM is its component name.

testmode output

INT .EXT

indicates if the I process is running in test mode. A nonzero value indicates yes; 0 indicates no (default).

Use the literal _COMPILED^IN^TESTMODE as the value for the testmode parameter. _COMPILED^IN^TESTMODE is automatically set to 1 if the source file is compiled with SETTOG 1, and 0 otherwise (indicating that the I process is running in production mode).

Test mode forces the STARTUP process parameter to default to yes, and enables processing of the CONFIG, STARTUP, and DEBUG process parameters (see Section 5, “DSNM Process Startup Functions”).

accept-startup-component output

INT .EXT

indicates whether a process COMPONENT value in the startup message should override (nonzero) or should not override (0) the *component* value. 0 (zero) is the default.

subject output

STRING .EXT

identifies the NULL-terminated value used in the EMS event messages. It can be the I process name.

Example

```
    INT PROC _STARTUP^MODE ( component, testmode,
             accept^startup^component, subject ) EXTENSIBLE;
    STRING .EXT component;
    INT    .EXT testmode;
    INT    .EXT accept^startup^component;
    STRING .EXT subject;

    BEGIN

      testmode       := _COMPILED^IN^TESTMODE;
      accept^startup^component  := 0;
      component ':=' [" "] & component FOR
$LEN(ZDSN^DDL^COMPONENT^DEF) - 1;
      component ':=' ["PATHWAY"];
      subject   ':=' ["PWI", 0];
      RETURN ZDSN^RETCODE^OK;
    END;
```

_ST^INITIAL

When the frame creates a thread, it sets the thread state to _ST^INITIAL. The thread state value is stored in an INT context variable, accessible with the _THREAD^STATE define.

See _ST^MIN^THREAD^STATE for information on defining your thread states.

```
_ST^INITIAL
```

Example

The following example tests the current state of the thread:

```
CASE _THREAD^STATE OF
  BEGIN
    _ST^INITIAL  ->
      ...
    OTHERWISE ->
      ...
  END;
  .
  .
  .
```

_ST^MIN^THREAD^STATE

_ST^MIN^THREAD^STATE is the minimum value of a user-defined thread state.

```
_ST^MIN^THREAD^STATE
```

Considerations

- Thread states are normally declared as literals.
- Values less than _ST^MIN^THREAD^STATE are reserved for use by the frame.
- _THREAD^STATE contains the current thread state.

Example

The following example declares several user-defined thread states and sets the current thread state:

```
LITERAL thr^state1 = _ST^MIN^THREAD^STATE, thr^state2,
        thr^state3,...;

_THREAD^STATE := thr^state2;
```

_SUBSYS^DEF

_SUBSYS^DEF is a template for a subsystem configuration structure, filled by the _ADD^SUBSYS procedure with subsystem and object-type configuration data. Declare an extended pointer to a _SUBSYS^DEF-defined subsystem configuration structure in globals for each subsystem your I process handles.

_ADD^SUBSYS must be called in your _STARTUP procedure for each subsystem your I process handles. _ADD^SUBSYS allocates the memory for, fills in, and returns the address of the subsystem configuration structure.

```
_SUBSYS^DEF
```

The definition of the _SUBSYS^DEF-defined structure is:

```
STRUCT _SUBSYS^DEF (*);
  BEGIN
    STRUCT SUBSYS^CONFIG (ZDSN^DDL^SUBSYS^CONFIG^DEF);
    INT OBJTYPES;
    STRUCT OBJTYPE^CONFIG (ZDSN^DDL^OBJTYPE^CONFIG^DEF)
      [0:-1];
  END;
```

_SUBSYS^DEF fills in one OBJTYPE^CONFIG array for each object type configured in the subsystem, and sets OBJTYPES to the number of object-type configuration entries in the subsystem.

Definitions for the two structures contained within the _SUBSYS^DEF-defined structure are as follows:

```
DEFINITION ZDSN-DDL-SUBSYS-CONFIG.
  02 Z-SUBSYS              TYPE ZDSN-DDL-SUBSYS.
  02 Z-RANK                TYPE ZSPI-DDL-INT.
  02 Z-DEFAULT-OBJTYPE     TYPE ZDSN-DDL-OBJTYPE.
  02 Z-DEFAULT-SUBOBJTYPE TYPE ZDSN-DDL-OBJTYPE.
  02 Z-DEVTYPE             OCCURS ZDSN-MAX-DEVTYPES TIMES.
    03 Z-TYPE                TYPE ZSPI-DDL-INT.
    03 Z-SUBTYPE             TYPE ZSPI-DDL-INT
                            OCCURS ZDSN-MAX-SUBTYPES TIMES.
  02 Z-FLAGS               TYPE ZSPI-DDL-ENUM.
  02 Z-MANAGER-OBJFILE     TYPE ZDSN-DDL-OBJNAME.
  02 Z-DSNMI               TYPE ZDSN-DDL-PCLASS.
END

DEFINITION ZDSN-DDL-OBJTYPE-CONFIG.
  02 Z-SUBSYS             TYPE ZDSN-DDL-SUBSYS.
  02 Z-OBJTYPE            TYPE ZDSN-DDL-OBJTYPE.
  02 Z-PARENT-OBJTYPE     TYPE ZDSN-DDL-OBJTYPE.
  02 Z-RANK               TYPE ZSPI-DDL-INT.
END
```

Example

```
< in global definitions >

INT .EXT ci^config (_CI^DEF);
INT .EXT ss^config (_SUBSYS^DEF);

STRING .ciname[0:ZDSN^MAX^CICLASS-1] := [“XXX ”];
STRING .ssname[0:ZDSN^MAX^SUBSYS-1] := [“YYYYYY ”];

< within _STARTUP procedure >

BEGIN
   .
   .
  IF _ISNULL (@ci^config := _ADD^CI (ciname)) THEN
    RETURN ZDSN^ERR^INTERNAL^ERR;
  IF _ISNULL (@ss^config := _ADD^SUBSYS (ssname)) THEN
   RETURN ZDSN^ERR^INTERNAL^ERR;
   .
   .
END;
```

_SUCCESSOR^LM

_SUCCESSOR^LM returns the address of the list member placed on the list immediately after the current list member was added.

```
@next-list-member := _SUCCESSOR^LM ( list
                                   ,list-member );
```

next-list-member returned value

INT .EXT

is the address of the successor list member.

list input

is the name of a _LIST.

list-member input

INT .EXT

is a pointer to a current member of *list*.

Considerations

- List members are logically ordered. The first (or front or head) member is the earliest put on the list and the last (or end or tail) member is the latest. Each member has a successor and a predecessor, the predecessor of the first and the successor of the last being _NULL.
- *list-member* must be a current member, or @*list-member* must be _NULL.

  If @*list-member* is _NULL, the address of the first member of *list* is returned.
- Successive list members are not necessarily stored at increasing memory addresses. You cannot determine the order of list members by comparing their addresses.
- _SUCCESSOR^LM returns _NULL if one of the following is true:
  - *list-member* is the last member of *list*.
  - *list* is empty.
  - An error occurs.

Examples

The following examples use the declarations:

```
_LIST (list);
INT .EXT lm (list^member^def);      !extended pointer to
                                    !list member structure
INT .EXT nextlm (list^member^def);  !another extended pointer
                                    !to list member struct
```

This example scans a list in the forward direction:

```
@lm := _NULL;
WHILE _NOTNULL (@lm := _SUCCESSOR^LM (list,lm)) DO
  BEGIN
  ...
  END;
```

In this example, the user waits for a new last member to be added to the end of a list by keeping a previous member pointer. After finding _NULL, @lm is set to its previous value. Later, @lm can be used in _SUCCESSOR^LM to get a new later member, if one has been added, or _NULL, if one has not been added.

```
@lm := @nextlm := _NULL;
WHILE _NOTNULL (@lm := _SUCCESSOR^LM (list,lm)) DO
  BEGIN
    @nextlm := @lm;
    ....
  END;
@lm := @nextlm;
     .
     .
  < lm can be used to find a new last member >
```

_THREAD^CONTEXT^ADDRESS

_THREAD^CONTEXT^ADDRESS is an INT(32) field, containing the extended address of the command context area, which is allocated to each thread when it is created and persists until the thread terminates. The context area contains a fixed header, followed by a user-defined area.

Since globals are shared among all threads, the construction to access the thread context must be done in the local data area of each procedure that requires access.

```
_THREAD^CONTEXT^ADDRESS
```

Example

The following example of a local data definition gives a thread procedure access to the command context area:

```
INT.EXT cx (command^context^def) = _THREAD^CONTEXT^ADDRESS;
```

`command^context^def` is a user-defined structure template that begins with _COMMAND^CONTEXT^HEADER.

See also the examples for _FOBJECT^INIT and _COMMAND^CONTEXT^HEADER.

_THREAD^PROC

_THREAD^PROC defines a procedure dispatched as part of a thread (a procedure that can be set as a new current thread with _SET^THREAD^PROC, _PUSH^THREAD^PROCSTATE, _POP^THREAD^PROCSTATE, _DISPATCH^THREAD or _SAVE^THREAD^AND^DISPATCH).

```
_THREAD^PROC ( procname );
```

procname user-provided identifier

is the name (a valid TAL identifier) of the procedure.

Examples

Use _THREAD^PROC in the following constructions:

```
_THREAD^PROC( procname ); EXTERNAL;

_THREAD^PROC( procname ); FORWARD;

_THREAD^PROC( procname );
  BEGIN
  < procedure body >
  _END^THREAD^PROC;
```

_THREAD^STATE

_THREAD^STATE accesses an INT variable that represents the current state of the thread. _THREAD^STATE may be set or tested.

```
_THREAD^STATE
```

Considerations

- The frame sets the thread state to _ST^INITIAL when it creates a thread. Subsequently, you may alter the thread state as desired; the frame never uses it again.
- Thread state values less than the library literal _ST^MIN^THREAD^STATE are reserved. State values are always nonnegative. At present, _ST^INITIAL is the only reserved value.
- The current thread state may altered with _DISPATCH^THREAD, _PUSH^THREAD^PROCSTATE, _POP^THREAD^PROCSTATE, _SAVE^THREAD^AND^DISPATCH, or _RESTORE^THREAD^AND^DISPATCH.

Example

The following example tests and alters the current state of the thread:

```
CASE _THREAD^STATE OF
  BEGIN
    _ST^INITIAL  ->
      ...
    OTHERWISE  ->
      ...
  END;
  .
  .
  .
_THREAD^STATE := thr^state2;
```

_THREAD^TERMINATION^CODE

_THREAD^TERMINATION^CODE is a define to access a context field that contains the ZDSN^ERR value returned with _RC^ABORT or _RC^STOP. It is designed for use in _COMMAND^TERMINATION^PROC to determine why the thread terminated.

```
_THREAD^TERMINATION^CODE
```

In the case of an _RC^STOP, the _THREAD^TERMINATION^CODE value is 0.

_THREAD^TERMINATION^PROC

_THREAD^TERMINATION^PROC defines a procedure responsible for cleaning up the thread's environment after a command successfully completes or after a thread abnormally terminates. It is the required name of the thread termination procedure for the I process.

_THREAD^TERMINATION^CODE may be accessed and/or altered in the thread termination procedure.

The thread termination procedure declared with _THREAD^TERMINATION^PROC must end with _END^THREAD^TERMINATION^PROC.

```
_THREAD^TERMINATION^PROC ( _COMMAND^TERMINATION^PROC);
```

Examples

Use _THREAD^TERMINATION^PROC in the following construction:

```
_THREAD^TERMINATION^PROC (_COMMAND^TERMINATION^PROC);
  BEGIN
     .
     .
  < procedure body >
     .
     ! For example, may free lists and return, leaving the
     ! thread's original termination code
     ! (_THREAD^TERMINATION^CODE) unchanged, and leaving
     ! official input and output lists to the frame.

     CALL _DEALLOCATE^LIST (...);
     CALL _CLOSE^CI (...);
     RETURN _RC^NULL;
  _END^THREAD^TERMINATION^PROC;
```

_TURNOFF

_TURNOFF turns off all the bits in *int-var* that are on in *bit-mask*.

```
_TURNOFF ( int-var , bit-mask );
```

int-var input/output

INT:ref

is a variable, the bits of which are turned off according to the contents of *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-var* to turn off.

Example

The following example turns off bits 9 and 11 in var:

```
INT var;
LITERAL evta = %20;           ! evta.<11> on
LITERAL evtb = %100;          ! evtb.<9> on
_TURNOFF (var, evta + evtb);  ! var.<9> and var.<11>
                              ! now off
```

_TURNON

_TURNON turns on all the bits in *int-var* that are on in *bit-mask*.

```
_TURNON ( int-var , bit-mask );
```

int-var input/output

INT:ref

is a variable, the bits of which are turned on according to the contents of *bit-mask*.

bit-mask input

INT:value

is an INT expression, the one-bits of which identify the bits in *int-var* to turn on.

Example

The following example turns on bits 9 and 11 in var:

```
INT var;
LITERAL evta = %20;          !evta.<11> on
LITERAL evtb = %100;         !evtb.<9> on
_TURNON (var, evta + evtb);  !var.<9> and var.<11> now on
```

_UNGET^LM

_UNGET^LM replaces the last list member removed from a list using _GET^LM.

```
error := _UNGET^LM ( list
                    ,list-member );
```

error returned value

INT

is a ZDSN^ERR value indicating the outcome of the call. See Appendix B, “DSNM Error Codes,” for error code definitions.

list input

is the name of a _LIST.

list-member input

INT .EXT

is a pointer to the member most-recently removed from *list* with _GET^LM.

Specifying a list member that is not the most-recently removed with _GET^LM invalidates the _UNGET^LM operation. Also, any intervening _PUT^LM or _DEALLOCATE^LIST invalidates the _UNGET^LM operation. Results from any of these are unpredictable.

Example

In the following example, members of `worklist` are examined and removed up to the first member that does not match a particular control value:

```
_LIST (worklist);
INT .EXT list^member (list^member^def);
                ! list^member^def includes control^field

WHILE _NOTNULL (@list^member := _GET^LM (worklist))
  AND list^member.control^field = current^ctl^value
    DO
      BEGIN
       < process all list^members matching current^ctl^value>
      END;

! Put non-matching list^member back
IF _NOTNULL (@list^member)
  THEN IF (error := _UNGET^LM (worklist, list^member))
    THEN ... < data corrupted > ;
```

_UNPOP^LM

_UNPOP^LM replaces the last list member removed from a list using _POP^LM.

```
error := _UNPOP^LM ( list
                   ,list-member );
```

error returned value

INT

is a ZDSN^ERR value, indicating the outcome of the call. See Appendix B, "DSNM Error Codes," for error code definitions.

list input

is the name of a _LIST.

list-member input

INT .EXT

is a pointer to the member most-recently removed from *list* with _POP^LM.

Specifying a list member that is not the most-recently removed with _POP^LM invalidates the _UNPOP^LM operation. Also, any intervening _PUT^LM or _DEALLOCATE^LIST invalidates the _UNPOP^LM operation. Results from any of these are unpredictable.

Example

In the following example, members of `worklist` are examined and removed up to the first member that does not match a particular control value:

```
_LIST (worklist);
INT .EXT list^member (list^member^def);
                ! list^member^def includes control^field

WHILE _NOTNULL (@list^member := _POP^LM (worklist))
  AND list^member.control^field = current^ctl^value
    DO
      BEGIN
       < process all list^members matching current^ctl^value>
      END;

! Put non-matching list^member back
IF _NOTNULL (@list^member)
  THEN IF (error := _UNPOP^LM (worklist, list^member))
    THEN ... < data corrupted > ;
```

_XADR^EQ

_XADR^EQ is a Boolean define statement that is TRUE if two possibly null extended addresses are equal (since _NULL can have more than one value).

```
_XADR^EQ ( address1
         , address2 )
```

address1 input

INT(32):value

is an extended address whose value is compared to *address2*.

address2 input

INT(32):value

is an extended address whose value is compared to *address1*.

_XADR^NEQ

_XADR^NEQ is a Boolean define statement that is TRUE if two possibly null addresses are not equal (since _NULL can have more than one value).

```
_XADR^NEQ ( address1
          , address2 )
```

address1 input

INT(32):value

is an extended address whose value is compared to *address2* .

address2 input

INT(32):value

is an extended address whose value is compared to *address1*.

B DSNM Error Codes

Scope of This Appendix

This appendix lists the ZDSN^ERR values that you may send back to the frame in the Z^RESULT field of a formatted output object structure, or that may be returned to you from a call to a DSNM library procedure.

Reporting Errors

Errors that do not terminate a command must be associated with an object (for instance, an object name that is unknown to the subsystem) and are reported in the Z^RESULT field of a formatted output object structure. In general, errors associated with a particular object should not terminate the command, although there may be exceptions for individual subsystems.

The Z^RESULT result code must be one of the ZDSN^ERR values defined in this appendix. In addition, the output object structure must contain all entries appropriate for the command being executed, including the fully qualified object name.

If the result code doesn't fully describe the error, additional descriptive information should be appended as result text (ZDSN^VTY^RESULTTEXT). The result text must not duplicate the information of the result code itself: presentation services substitute the text below for the result code in the user's error display.

What to Prepare Before Contacting Your Tandem Support Representative

Some of the problems you encounter might require assistance from a Tandem support representative. Before you contact your representative, gather the following relevant information:

- How DSNM is installed.
- Whether DSNM was started before starting NonStop NET/MASTER MS, or NonStop NET/MASTER MS started DSNM.
- The VPROC of any utility executed by PROGRUN or OPSYS (with NonStop NET/MASTER MS environments).
- History of the problem: has it happened before? If so, when and under what conditions did it happen? Can you reproduce it? If so, state how to reproduce it.
- List of any recent changes to the system, including, but not limited to, new or changed configuration parameters, new or upgraded software modules, new hardware components, or changed NCL procedures (with NonStop NET/MASTER MS environments).
- List of the output of any tracing associated with the problem, if appropriate to the environment or nature of the problem.

- List of any warning or error messages displayed before and after the problem occurred.
- Any other information, as stated in the explanation of the pertinent error messages.

ZDSN Error Codes

The following ZDSN^ERR values may be returned to the frame by the command thread in the Z^RESULT field of a formatted output object, or returned to the command thread from a call to a library procedure. Error numbers -1 through -29 are standard SPI error codes; refer to the *SPI Programming Manual* for information on SPI error codes.

-nnn

```
Unexpected Error: text
```

Cause. There is an internal problem in the software that issued the message.

Effect. The effects of this problem vary, depending on the individual situation.

Recovery. Note the error number and the message text and contact a Tandem representative. Prepare the necessary information as suggested in “What to Prepare Before Contacting Your Tandem Support Representative” on page B-1.

0 ZDSN^ERR^NOERR

Cause. This is not an error condition.

Effect. The operation completed successfully.

Recovery. None.

-30 ZDSN^ERR^CMD^MISMATCH

```
Invalid Command For This Object
```

Cause. You issued a command against objects for which the command is not allowed.

Effect. Your command is not executed on the objects for which it is invalid.

Recovery. Refer to the *User’s Guide to DSNM Commands* for information on which commands are valid for the object type you specified.

-34 ZDSN^ERR^INTERNAL^ERR

```
Unexpected Error: DSNM Component Error
```

Cause. An internal program error or inconsistency occurred.

Effect. Command terminated abnormally due to an internal DSNM component error, possibly due to memory corruption.

Recovery. Check the EMS event log for the error reported. Stop the I process and then start it again. If the error persists, contact your Tandem representative.

-35 ZDSN^ERR^SUBSYSTEM^ERR

```
Subsystem Error
```

Cause. In attempting to execute your command, the subsystem generated an error. The name of the subsystem appears in the error message.

Effect. The command is executed on all objects except the one for which the error occurred. The state of the error object is subsystem-dependent.

Recovery. Refer to the subsystem documentation for error information.

-44 ZDSN^ERR^TKN^VAL^INV

```
Invalid Token or Operand Value
```

Cause. Your command included something invalid in the position where a keyword or operand is expected.

Effect. Your command is not executed.

Recovery. See Section 2, “DSNM Commands,” for the correct syntax of the command and reissue it.

-45 ZDSN^ERR^TKN^REQ

```
Required Token or Operand Missing
```

Cause. Your command omitted something that is syntactically required.

Effect. Your command is not executed.

Recovery. See Section 2, “DSNM Commands,” for the correct syntax of the command and reissue it.

-51 ZDSN^ERR^SPI^ERR

```
Unexpected Error: DSNM SPI Error
```

Cause. A subsystem SPI error occurred. Append the SPI error to the output object as ZDSN^VTY^RESULTTEXT.

Effect. The command terminated abnormally. The state of the error object is subsystem-dependent.

Recovery. Check the CI message SPI buffer for correctness. Refer to the subsystem management programming documentation for information about the error.

-55 ZDSN^ERR^OBJNAME^INV

```
Invalid Object Name
```

Cause. You specified a syntactically invalid object name.

Effect. Your command is not executed on the objects with the invalid name(s).

Recovery. Reissue the command with the correct object name.

-56 ZDSN^ERR^OBJTYPE^NOT^SUPPORTED or ZDSN^ERR^OBJ^NOT^SUPP

```
Object Type not Supported
```

Cause. You specified an object type that is not supported by DSNM. It is possible for DSNM to support a subsystem without supporting all of its object types.

Effect. Your command is not executed on objects of the unsupported type.

Recovery. None.

-60 ZDSN^ERR^MEMORY or ZDSN^ERR^NO^MEM^SPACE

```
Out of Memory
```

Cause. There is insufficient memory to execute your command.

Effect. Your command might have been partially executed. Use the information commands to determine to what extent the command was executed.

Recovery. Configure a larger segment size or break the command into several smaller commands. For information on configuring a larger segment size, refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide*.

-64 ZDSN^ERR^FS^ERR

```
File System Error
```

Cause. The Guardian file system generated an error during the execution of your command.

Effect. The effect depends on the specific file system error.

Recovery. Refer to the *Guardian User's Guide* for an explanation of the error and its recovery.

-67 ZDSN^ERR^CMD^TIMED^OUT

```
Command Timeout
```

Cause. The command timed out before it could be executed.

Effect. Your command is not executed.

Recovery. Issue the command again.

-69 ZDSN^ERR^CMD^NOT^SUPP

```
Command not Supported
```

Cause. You issued a command that is not supported in the specified subsystem or on the specified object types.

Effect. Your command did not affect the objects for which the command is not supported.

Recovery. Refer to the *User's Guide to DSNM Commands* for information on which commands are supported for the subsystem and objects you specified.

-71 ZDSN^ERR^ALLOCATESEGMENT^ERR

```
Segment Allocation Error
```

Cause. There was a segment allocation error during the execution of your command.

Effect. Your command is not executed.

Recovery. If there is not enough disk space, configure the SWAPVOL disk for more space for the segment swap file. Refer to the *Distributed Systems Management Solutions (DSMS) System Management Guide* for information.

-76 ZDSN^ERR^BADCOMMAND

```
Unexpected Error: Invalid Command
```

Cause. The command is not valid for the subsystem.

Effect. The command is not executed.

Recovery. Refer to the *User's Guide to DSNM Commands* for information on which commands are supported for the subsystem you specified.

-77 ZDSN^ERR^UNSUPPORTED^BY^SUBSYS

```
Not Supported by Subsystem
```

Cause. The operation or command modifier is not supported by the subsystem you specified.

Effect. Your command is not executed.

Recovery. Refer to the *User's Guide to DSNM Commands* for information about the commands supported by the specified subsystem.

-78 ZDSN^ERR^UNSUPPORTED^BY^I

```
Not Supported by DSNM Interface
```

Cause. You attempted to use an operation or command modifier that is not supported by the DSNM interface.

Effect. Your command is not executed.

Recovery. Refer to the *User's Guide to DSNM Commands* for information about the commands you can use.

-79 ZDSN^ERR^DATA^INTEGRITY

```
Unexpected Error: Data Integrity Error
```

Cause. Arguments are inconsistent or data structures have been corrupted.

Effect. The frame waits for the next event to redispatch the command thread. The command may yield erroneous results.

Recovery. Stop and restart the I process. If the error persists, contact your Tandem representative.

-81 ZDSN^ERR^MISSING^OBJTYPE

```
Missing Object Type
```

Cause. You issued a command without specifying the object type, and the object type could not be determined from the information on the command line.

Effect. Your command is not executed on the affected objects.

Recovery. Reissue the command, specifying the object type explicitly.

-82 ZDSN^ERR^BADOBJTYPE

```
Invalid Object Type
```

Cause. You issued a command, specifying an object type that is not valid for the subsystem.

Effect. Your command is not executed on the affected objects.

Recovery. Refer to the *User's Guide to DSNM Commands* for information about valid object types for the subsystem.

-86 ZDSN^ERR^REQ^KEYWORD^MISSING

```
Required Keyword Missing
```

Cause. The command has a required keyword missing.

Effect. The command is not executed.

Recovery. Refer to the *User's Guide to DSNM Commands* for information about command syntax and required keywords.

-88 ZDSN^ERR^DUP^KEYWORD

```
Duplicate keyword
```

Cause. A keyword is repeated in the command.

Effect. The command is not executed.

Recovery. Refer to the *User's Guide to DSNM Commands* for information about command syntax.

-202 ZDSN^ERR^OBJECTTOOLONG or ZDSN^ERR^OBJTOOLONG

```
Object Name too Long
```

Cause. You typed an object name that is longer than the maximum allowable length.

Effect. Your command is not executed.

Recovery. Reissue the command with a shorter name.

-204 ZDSN^ERR^BADARGUMENT

```
Missing or Invalid Library Argument
```

Cause. There is an internal problem in the software that issued the message.

Effect. The effects of this problem vary, depending on the situation.

Recovery. Contact a Tandem representative. Prepare the necessary information as suggested in “What to Prepare Before Contacting Your Tandem Support Representative” on page B-1.

-206 ZDSN^ERR^NOTPUSHED

```
Thread Proc was not pushed
```

Cause. _POP^THREAD^PROCSTATE or _RESTORE^THREAD^AND^DISPATCH was executed without a procedure having first been placed o n the stack.

Effect. Thread procedure not executed, and the command terminated abnormally.

Recovery. Check the source code to ensure that the thread procedure is saved before it is dispatched again. If necessary, correct the source code.

-207 ZDSN^ERR^LIB^BADVALUE^OMITTED

```
Invalid DSNM configuration parameter values
```

Cause. One or more DSNM configuration parameter records contained an invalid numeric value.

Effect. The library procedure reporting this error substitutes a value of -1 for the erroneous value in the affected output parameter.

Recovery. You may be able to determine which value was unacceptable by looking for a -1 returned in some field for which a positive value was expected. Using NETCOM, check the DSNM configuration records in your DSNM configuration file, if any, and correct the record.

-212 ZDSN^ERR^SYNTAX

```
Invalid Syntax
```

Cause. Your command was syntactically incorrect.

Effect. Your command is not executed.

Recovery. Verify the syntax of the command, correct it as needed, and reissue the command. See Section 2, “DSNM Commands,” for information.

-214 ZDSN^ERR^RESERVEDWORD

```
Reserved Word Misplaced
```

Cause. Your command included a reserved word in the wrong position.

Effect. Your command is not executed.

Recovery. Verify the syntax of the command, correct it as needed, and reissue the command. See Section 2, “DSNM Commands,” for further information.

-216 ZDSN^ERR^CMDERROR

```
Unexpected Error: Invalid Command or Option
```

Cause. Your command or one of its options was not valid.

Effect. Your command is not executed.

Recovery. Verify the syntax of the command, correct it as needed, and reissue the command. See Section 2, “DSNM Commands,” for further information.

-217 DSN^ERR^BADLOGON

```
Invalid Logon Info
```

Cause. The software from which you attempted to use DSNM may be incorrectly configured. DSNM was invoked with incorrect data.

Effect. DSNM is not invoked.

Recovery. Notify your system manager.

Messages From the DSNM Parser

The following errors may be generated by the DSNM parser, which interprets DSNM commands before they are executed.

```
Command not recognized
```

Cause. You misspelled a command or issued a command that is not supported.

Effect. Your command is not executed.

Recovery. Correct and reissue the command.

```
Exceeded max objects
```

Cause. Your command includes more objects than are allowed in a single command.

Effect. Your command is not executed.

Recovery. Simplify the command, or break it into multiple commands if necessary.

```
Exceeded max paren levels
```

Cause. You nested parentheses beyond the maximum allowable depth.

Effect. Your command is not executed.

Recovery. Simplify the command or break it into two or more commands, if necessary.

```
Exceeded param space
```

Cause. You entered too many parameters or an excessively long parameter.

Effect. Your command is not executed.

Recovery. Simplify the command, or break it into multiple commands if necessary.

```
Invalid Error number:  error
```

Cause. An error that DSNM does not recognize occurred during the processing of your command. This is the mechanism by which file system errors associated with Tandem data communications subsystems other than AM3270, Expand, SNAX/CDF, SNAX/XF, TR3271, or X25AM are reported.

Effect. The effects of this problem vary, depending on the situation.

Recovery. If the reporting subsystem is a Tandem data communications subsystem other than AM3270, Expand, SNAX/XF, SNAX/CDF, TR3271, or X25AM, and the

error is a positive number, refer to the *Guardian Procedure Errors and Messages Manual* for an explanation of the file system error and its recovery.

If the error is not a file system error, call your Tandem representative. Prepare the necessary information as suggested in "What to Prepare Before Contacting Your Tandem Support Representative" on page B-1.

```
Invalid Object Type
```

Cause. You specified an object type that is not valid.

Effect. Your command is not executed.

Recovery. Correct and reissue the command.

```
Invalid option
```

Cause. You specified a modifier or parameter that is not valid for any command.

Effect. Your command is not executed.

Recovery. Correct and reissue the command.

```
Invalid option for this command
```

Cause. A modifier or parameter was entered that is not valid with this command.

Effect. Your command is not executed.

Recovery. Correct and reissue the command.

```
Missing Object Type
```

Cause. You omitted the object type when it was required.

Effect. Your command is not executed.

Recovery. Correct and reissue the command.

```
Name too long
```

Cause. You entered a name that is longer than the maximum legal length.

Effect. Your command is not executed.

Recovery. Correct the name and reissue the command.

```
No operands
```

Cause. You issued a command that requires operands, but did not specify any.

Effect. Your command is not executed.

Recovery. Correct and reissue the command.

```
Reserved word misplaced
```

Cause. A keyword, subsystem name, or object type was out of place.

Effect. Your command is not executed.

Recovery. Correct and reissue the command. If the error was caused because an object name is the same as a keyword, subsystem name, or object type, enclose the object name in quotation marks.

```
Syntax error
```

Cause. The command contains a serious syntax error.

Effect. Your command is not executed.

Recovery. Correct and reissue the command.

```
Unbalanced parens
```

Cause. Your command includes parentheses that are incorrectly paired.

Effect. Your command is not executed.

Recovery. Correct and reissue the command.

```
Unexpected end
```

Cause. The command did not include all required and expected information.

Effect. Your command is not executed.

Recovery. Correct and reissue the command.

```
Unexpected error: text
```

Cause. There is an internal problem in the software that issued the message.

Effect. The effects of this problem vary, depending on the situation.

Recovery. Note the error text and contact a Tandem representative. Prepare the necessary information as suggested in “What to Prepare Before Contacting Your Tandem Support Representative” on page B-1.

Defined DSNM SPI Components

Scope of This Appendix

Internally, DSNM uses the Tandem Subsystem Programmatic Interface (SPI) for command and response message flows. DSNM SPI components (constants and data definitions) are defined in the DSNM SPI Data Definition Language (DDL) and may be identified by the prefix ZDSN^.

SPI messages are handled by the frame and are generally hidden from the user-written command thread, but certain SPI DDL constants and structure definitions are required. This appendix lists the SPI DDL constants and structure definitions that user-written procedures must use.

Commands

The following action codes identify the DSNM commands:

ZDSN^ACTION^ABORT
ZDSN^ACTION^AGGREGATE
ZDSN^ACTION^INFO
ZDSN^ACTION^START
ZDSN^ACTION^STATISTICS
ZDSN^ACTION^STATUS
ZDSN^ACTION^STOP

Modifiers

The following structure definition defines the modifiers in a DSNM command. STRUCT ZDSN^DDL^MOD^DEF contains the following fields of interest:

INT Z^AMOD
INT Z^EMOD
INT Z^HMOD
INT Z^RMOD
INT Z^SMOD

HMOD Values

The following constants are the possible Z^HMOD values:

Zero (omitted)—Default to ZDSN^HMOD^ALL
ZDSN^HMOD^ALL
ZDSN^HMOD^ONLY
ZDSN^HMOD^SUBONLY

EMOD Values

The following constants are the possible Z^EMOD values:

Zero (omitted)—Default to ZDSN^EMOD^BRIEF
ZDSN^EMOD^BRIEF
ZDSN^EMOD^DETAIL
ZDSN^EMOD^SUPPRESS

SMOD Values

The following constants are the possible Z^SMOD values:

0 (omitted)—No default—Command applied regardless of object state
ZDSN^SMOD^GREEN / ZDSN^SMOD^UP
ZDSN^SMOD^NOT^GREEN / ZDSN^SMOD^NOT^UP
ZDSN^SMOD^RED / ZDSN^SMOD^DOWN
ZDSN^SMOD^NOT^RED / ZDSN^SMOD^NOT^DOWN

ZDSN^SMOD^GREEN and ZDSN^SMOD^UP have the same value and may be used interchangeably. Similarly NOT^GREEN/NOT^UP, RED/DOWN, and NOT^RED/NOT^DOWN are interchangeable.

RMOD Values

The following constants are the possible Z^RMOD values:

Zero (omitted)—Default to ZDSN^RMOD^BRIEF
ZDSN^RMOD^BRIEF
ZDSN^RMOD^DETAIL
ZDSN^RMOD^SUMMARY
ZDSN^RMOD^SUMMARY^BYCOMPONENT
ZDSN^RMOD^SUMMARY^BYNAME
ZDSN^RMOD^SUMMARY^BYOBJECT
ZDSN^RMOD^SUMMARY^BYTYPE

AMOD Values

The following constants are the possible Z^AMOD values:

Zero (omitted)—Default; do not reset statistics
ZDSN^AMOD^CANCEL
ZDSN^AMOD^RESET

Command Object DDL

The following structure definition defines an object in a DSNM command. STRUCT ZDSN^DDL^FOBJECT^DEF contains the following fields of interest:

INT Z^RESULT

INT Z^HMOD = Z^RESULT

STRUCT Z^SUBSYS (ZDSN^DDL^SUBSYS^DEF)

STRUCT Z^OBJTYPE (ZDSN^DDL^OBJTYPE^DEF)

INT Z^OBJNAME^OCCURS

STRUCT Z^OBJNAME (ZDSN^DDL^OBJNAME^DEF)

INT Z^MANAGER^OCCURS

STRUCT Z^MANAGER (ZDSN^DDL^MANAGER^DEF)

An object in a command contains an HMOD in the Z^HMOD field. In the response, the Z^RESULT field (which redefines Z^HMOD) contains a state (for the STATUS command) or a ZDSN^ERR error code.

Note. The ZDSN^ERR value ZDSN^ERR^NOERR (value zero) is the no-error value for all responses except for the STATUS command.

DSNM State Values

The following constants are the DSNM object state values:

ZDSN^STATE^GREEN / ZDSN^STATE^UP
ZDSN^STATE^RED / ZDSN^STATE^DOWN
ZDSN^STATE^YELLOW / ZDSN^STATE^PENDING
ZDSN^STATE^NULL
ZDSN^STATE^UNKNOWN
ZDSN^STATE^UNDEFINED

ZDSN^STATE^GREEN and ZDSN^STATE^UP are interchangeable, as are RED/DOWN and YELLOW/PENDING.

Error Codes

The following constants are the DSNM error codes used most often:

```
ZDSN^ERR^NOERR
ZDSN^ERR^INTERNAL^ERR
ZDSN^ERR^SUBSYSTEM^ERR
ZDSN^ERR^OBJNAME^INV
ZDSN^ERR^OBJTYPE^NOT^SUPPORTED | ZDSN^ERR^OBJ^NOT^SUPP
ZDSN^ERR^MEMORY
ZDSN^ERR^FS^ERR
ZDSN^ERR^CMD^NOT^SUPP
ZDSN^ERR^UNSUPPORTED^BY^SUBSYS
ZDSN^ERR^UNSUPPORTED^BY^I
ZDSN^ERR^MISSING^OBJTYPE
```

AGGREGATE Counters

The following structure definition defines counters returned in an AGGREGATE command response. STRUCT ZDSN^DDL^COUNTERS^DEF contains the following fields of interest:

```
INT(32)  Z^GREEN;
INT(32)  Z^UP = Z^GREEN;
INT(32)  Z^RED;
INT(32)  Z^DOWN = Z^RED;
INT(32)  Z^YELLOW;
INT(32)  Z^PENDING = Z^YELLOW;
INT(32)  Z^UNDEFINED;
INT(32)  Z^INERROR;
```

Response Item Types

Command responses often require items to be appended to the response object. The following constants define the types of items that may be appended:

```
ZDSN^VTY^COUNTERS
ZDSN^VTY^ERRORTEXT
ZDSN^VTY^NONTEXT
ZDSN^VTY^RESULTTEXT
ZDSN^VTY^TEXT
```

The maximum length of a text line (RESULTTEXT, TEXT, or ERRORTEXT) that may be appended is the DDL constant ZDSN^MAX^TEXT (75 characters).

DDL Definitions for DSNM Character String Components

Subsystems and object types in DSNM commands, responses, and configuration are represented by character strings. In TAL, the representation of a character-string type item has the following form:

```
LITERAL zdsn^max^item = length-of-item-in-bytes;

STRUCT zdsn^ddl^item^def;
  BEGIN
    STRING z^c[0:zdsn^max^item-1];
    INT z^i = z^c;
  END;
```

All character-string type items are defined this way: a constant (with the item's length in bytes) and a uniform structure that allows it to be referred to as a structure, a string, or an INT.

Note. If one of these items is embedded in another structure, care must be taken to ensure that the structure begins on a word boundary.

The major character-string items of interest are described by the following DDL items:

Item Describes	DDL Name	Length
System name	ZDSN^DDL^SYSTEM^DEF	ZDSN^MAX^SYSTEM
Subsystem name	ZDSN^DDL^SUBSYS^DEF	ZDSN^MAX^SUBSYS
Object type	ZDSN^DDL^OBJTYPE^DEF	ZDSN^MAX^OBJTYPE
Object name	ZDSN^DDL^OBJNAME^DEF	ZDSN^MAX^OBJNAME
Manager name	ZDSN^DDL^MANAGER^DEF	ZDSN^MAX^MANAGER
Process name	ZDSN^DDL^PNAME^DEF	ZDSN^MAX^PNAME
Process qualifier	ZDSN^DDL^PQUAL^DEF	ZDSN^MAX^PQUAL
Process class	ZDSN^DDL^PCLASS^DEF	ZDSN^MAX^PCLASS
CI class	ZDSN^DDL^CICLASS^DEF	ZDSN^MAX^CICLASS

Note. A CI class is an instance of a process class (pclass).

DSNM Configuration Items

Item Describes	DDL Name	Length
Class	ZDSN^DDL^CLASS^DEF	ZDSN^MAX^CLASS
Component	ZDSN^DDL^COMPONENT^DEF	ZDSN^MAX^COMPONENT
Parameter name	ZDSN^DDL^PARAMNAME^DEF	ZDSN^MAX^PARAMNAME
Parameter value	ZDSN^DDL^CONFPARAMVALUE ^DEF	ZDSN^MAX^CONFPARAM VALUE

D Sample I Process Program Code

Scope of This Appendix

Appendix D provides a sample I process program for a pseudo-subsystem (SPIFFY), illustrating the program model and associated I process development library services described in this manual.

Overview of the SPIFFY Subsystem

The SPIFFY subsystem consists of SPIFFY manager processes, each controlling a set of SPIFFY objects. A manager supports a programmatic interface consisting of formatted messages. There are seven object types arranged in the following hierarchy:

The hierarchical parent of a SPIFFY object is known as the object's "Pop."

Characteristics of SPIFFY Objects

A REACTOR object comprises an internal group of objects and has no individual characteristics of its own. The other objects have various individual characteristics, as follows:

- BOILER, VALVE, and CHAMBER objects consist of a number of elements (each of which is too insignificant to be considered a separate entity). When these elements are raised to a temperature above absolute 0, they exert pressure. BOILER, VALVE, and CHAMBER have the following characteristics:
  - Pressure
  - Volume
  - Temperature
  - Number of elements
- ASSEMBLY, COGWHEEL, and GEAR objects have the following characteristics
  - Color: YELLOW, CYAN, MAGENTA, BLACK
  - Composition: IRON, STEEL, COPPER, BRASS
  - State for ASSEMBLY objects: STOPPED, GOING
  - State for COGWHEEL and GEAR objects: IDLE, COASTING, ROTATING, LOCKED

    An object is LOCKED when its "Pop" object is STOPPED.

SPIFFY Subsystem Programmatic Interface Commands

The SPIFFY subsystem programmatic interface supports informational and state-change commands.

Informational Commands

The SPIFFY subsystem programmatic interface supports one informational command, TELLABOUT, which applies to objects of all types and returns everything known about the object(s) specified by the command. TELLABOUT has the following operands:

```
NAME { objname | * }
POP { objname | * }
TYPE { typecode | ANYTHING }
```

Multiple objects can be specified in a TELLABOUT command by supplying an asterisk (*) as the object name (NAME) or parent name (POP), and the ANYTHING type code to specify objects of any type. (The asterisk allows you to specify objects and parents of any name.)

NAME, POP, and TYPE may be used in the following combinations to retrieve information about objects:

NAME	POP	TYPE	Returns
Objname1	Objname2	*typecode*	Object Objname1 of type *typecode* with Pop Objname2
*	Objname2	*typecode*	All objects of type *typecode* with Pop Objname2
Objname1	*	*typecode*	All objects Objname1 of type *typecode* with any Pop
*	*	*typecode*	All objects of type *typecode*
Objname1	Objname2	ANYTHING	Object Objname1 of any type with Pop Objname2
*	Objname2	ANYTHING	All objects of any type with Pop Objname2
Objname1	*	ANYTHING	All objects Objname1 of any type with any Pop
*	*	ANYTHING	All objects

State-Change Commands

State-change commands behave differently for different object types. There are no state-change commands for the REACTOR object. For the other objects, the following operations are supported:

BOILER, VALVE, CHAMBER	WARMUP, HEATUP, COOLOFF, SHUTOFF
ASSEMBLY	GO, DONTGO
COGWHEEL, GEAR	LOOSENUP, SPEEDUP, SLOWUP, LOCKUP

State-change commands do not support the asterisk (*) for NAME, nor do they support the ANYTHING type code.

The program code example in this appendix illustrates the processing of informational commands. The processing of state-change operations is not explicitly illustrated; however, the preliminary processing of NAME, POP, and TYPE values is included (to determine the set of objects on which a state-change operation would be performed).

Command and Response Message Formats

A SPIFFY command is executed by sending a command message to a SPIFFY manager process. The results of the command message are returned in a response message. A single command may consist of several command-to-response exchanges.

Command Message Format

The command message has the same format for all commands and all SPIFFY object types:

```
LITERAL spiffy^name^len= 20;

STRUCT spiffy^command^def (*); ! Command message, which also
                               ! heads response message
  BEGIN
   INT cmd;
   INT type;
   INT response^context;
   STRING name[0:spiffy^name^len-1];
   STRING pop^name[0:spiffy^name^len-1];
  END;
```

The command message contains the following:

- A command code (CMD)
- A object type code to which the command applies (TYPE)
- The name and parent of the object to which the command applies (NAME and POP^NAME)

RESPONSE^CONTEXT must be set to 0 when a command is originated.

Response Message Format

The response message begins with the command message, followed by an error code (ERROR), a count of response objects (RESPONSE^THINGS), and response object details in an object-characteristics array (THING):

```
LITERAL max^response^things = 2;

STRUCT spiffy^response^def (*); ! Response message, which
                                ! contains cmd msg struct
  BEGIN
   STRUCT cmd (spiffy^command^def);
   INT error;
   INT response^things;
   STRUCT thing (spiffy^thing^def) [0:max^response^things-1];
  END;
```

If an error occurs, an error code is returned in the first response; see "SPIFFY Subsystem Literal Definitions" later in this section. In this case, there are no response objects. If the command succeeds, the code COMMAND^DONE (0) is returned, along with response object(s).

If a TELLABOUT command specifies more objects than can be returned in a single response, the RESPONSE^CONTEXT field in the response message CMD structure is set to a nonzero value. In this case, returning the CMD structure exactly as it appears in the response message causes the next set of response objects to be returned. This process is repeated until the response message CMD.RESPONSE^CONTEXT is 0, indicating the return of all objects.

The THING structure returns the characteristics of SPIFFY objects and is the same for all object types:

```
 STRUCT spiffy^thing^def (*);
  BEGIN
   INT type;                           ! object type
   INT name^occurs;                    ! length of object name
   STRING name [0:spiffy^name^len-1]; ! object name
   INT pop^name^occurs;                ! length of name of
                                       ! object's parent
   STRING pop^name[0:spiffy^name^len-1];  ! parent name
   ! For ASSEMBLYs and subordinates
   INT state;                          ! ASSEMBLY state
   INT color;                          ! and other essentials
   INT composition;
   ! For the various REACTOR components
   INT(32) temp;                       ! Degrees Kelvin
   INT(32) press;                      ! MM Hg
   INT(32) vol;                        ! Liters
   INT(32) n;                          ! Number of elements
  END;
```

SPIFFY Subsystem Literal Definitions

Literal definitions for all codes and errors are provided when the SPIFFY subsystem is delivered.

Codes: Object Types, States, Operations, Characteristics

```
LITERAL
  ! Types of objects
  LEAST^LITTLE^THING, REACTOR = LEAST^LITTLE^THING,
  BOILER, VALVE, CHAMBER, ASSEMBLY, COGWHEEL, GEAR, OTHER,
  ALMOST^ANYTHING = OTHER, ANYTHING,

  ! States of objects
  STOPPED, GOING,                          ! ASSEMBLY states
  LOCKED, ROTATING, COASTING, IDLE,        ! COGWHEEL and GEAR
                                           ! states

  ! Operations on objects
  TELLABOUT,                               ! Operations for
                                           ! all objects
  WARMUP, HEATUP, COOLOFF, SHUTOFF,        ! REACTOR component
                                           ! operations
  GO, DONTGO,                              ! ASSEMBLY operations
  LOOSENUP, SPEEDUP, SLOWUP, LOCKUP,       ! COGWHEEL and GEAR
                                           ! operations

  ! Characteristics of objects
  BLACK, YELLOW, CYAN, MAGENTA,            ! ASSEMBLY, COGWHEEL,
                                           ! GEAR in 4 colors
  IRON, STEEL, COPPER, BRASS;              ! made 4 ways
```

Errors

```
LITERAL COMMAND^DONE, COMMAND^MSG^TOOSHORT, COMMAND^INVALID,
        TYPE^INVALID, COMMAND^INVALID^FOR^TYPE,
        COMMAND^IMPOSSIBLE, THING^NONEXISTENT,
        POP^NONEXISTENT;
```

SPIFFY I Process Design

To develop an I process, you must map the states of subsystem objects to DSNM states, and a sequence of subsystem commands to the DSNM commands.

State Mapping

To map the many possible states of subsystem objects to DSNM states requires detailed subsystem knowledge. The mapping should represent the operating state of the subsystem object. The DSNM states GREEN, RED, YELLOW represent good, poor, and transitional or borderline operational health. Suppose you enlisted the aid of a SPIFFY subsystem expert, who tells you that:

- REACTOR objects cannot have a state; they only exist (or not).
- BOILER, VALVE, and CHAMBER operation depends only on temperature, which must be neither too high nor too low.
- ASSEMBLY, COGWHEEL, and GEAR operation depends only on the SPIFFY state at the moment.

The following state mapping is assigned:

Object Type		Operating Condition	DSNM State Assigned
REACTOR		N.A.	NULL
	Temperature		
BOILER /	0-297	Too low to operate	RED
VALVE /	298-595	Low but operable	YELLOW
CHAMBER	596-893	Optimum operating range	GREEN
	894-1191	High but operable	YELLOW
	1192-up	Too high to operate	RED
	State		
ASSEMBLY	STOPPED	Inoperable	RED
	GOING	Operable	GREEN
COGWHEEL /	LOCKED	Unusable	RED
GEAR	IDLE	Inoperable	RED
	COASTING	Marginally operable	YELLOW
	ROTATING	Operable	GREEN

Note. Within a subsystem, it is possible for more than one set of conditions to map to the same DSNM state. Also, not all DSNM states need be present for every object type.

Implementing DSNM Commands

DSNM commands are implemented by SPIFFY subsystem commands as follows.

Implementing the Informational Commands

The DSNM STATUS and INFO commands can be issued by executing the TELLABOUT command and selecting different informational details to return in the output. The SPIFFY subsystem does not support a STATISTICS command or its equivalent.

DSNM Command	SPIFFY Command	Comment
INFO	TELLABOUT	Returns selected information fields according to object type.
STATUS	TELLABOUT	Returns selected status information.
STATISTICS	N.A.	Returns "No STATISTICS Available" (see following note).
AGGREGATE	TELLABOUT	Combines the results of: NAME * POP * TYPE ANYTHING by object type.

Note. How you handle a DSNM command when no subsystem-equivalent command exists depends on what is operationally reasonable and how much information you want to return. For example, with the STATISTICS command here, you could return ZDSN^ERR^UNSUPPORTED^BY^SUBSYS or ZDSN^ERR^NOERR, or possibly have the I process keep its own statistics, thus simulating the operation.

Implementing the Hierarchy Modifiers

Assuming that object names are unique under a given manager, the various combinations of "*" names and the ANYTHING type code listed next can be used to construct all of the possible DSNM hierarchy modifiers.

	TELLABOUT Command with		
HMOD	**NAME**	**POP**	**TYPE**
ONLY	*objname*	*	*typecode*
SUBONLY	*	*objname*	ANYTHING
ALL followed by	*objname* *	* *objname*	*typecode* ANYTHING

Implementing the State Modifiers

There is no SPIFFY equivalent of SMOD. The only way to implement something analogous is by using the TELLABOUT command and selecting those objects whose DSNM mapped state satisfies the DSNM SMOD.

Implementing an Informational Command on a "*" Operand

You can implement an informational command on a "*" operand by first issuing a TELLABOUT command:

```
TELLABOUT NAME * POP * TYPE typecode
```

This ignores HMOD and SMOD. Perform the informational operation on each resulting object, as described in the hierarchy modifier table above.

Implementing State-Change Commands

Since the SPIFFY subsystem state-change commands do not support a "*" objname for NAME, such commands usually require that a TELLABOUT command be issued first to determine the list of objects on which to carry out the state-change operation. Essentially, this is an internal DSNM STATUS command (applying modifiers), followed by a state-change command for each resulting object.

Managing SPIFFY Through DSNM: Sample Command Output

The following examples illustrate how to test the DSNM STATUS command on various SPIFFY subsystem objects, using DSNMCom to send commands to the SPIFFY I process.

Using DSNMCom to Test the SPIFFY I Process

If they are not already running, start the SPIFFY I process and the SPIFFY manager process(es). For example:

```
> RUN $DSNM.IDEV.SPIFI/NAME $SPFI,NOWAIT/TESTMODE 1 &
     CONFIG $DSNM.IDEV.DSNMCONF

> RUN $DSNM.IDEV.SPIFMGR/NAME $SMGR,NOWAIT/
```

See "Configuring SPIFFY Into DSNM" on page D-28 for a description of the DSNMCONF file referred to in this example. See also Section 5, "DSNM Process Startup Functions," for information about DSNM process startup parameters.

Next start DSNMCom, opening the SPIFFY I process and specifying the configuration file into which the SPIFFY subsystem records have been added. For example:

```
> DSNMCOM CONFIG $DSNM.IDEV.DSNMCONF
DSNMCom - T9216D30 12FEB95
Copyright Tandem Computers Incorporated 1995
DSNMCom> open $spfi
DSNM $spfi >
```

See Section 7, "DSNMCom: The I Process Test Utility," for a description of DSNMCom.

DSNM STATUS Command Output

Following are examples of STATUS command output:

```
DSNM $spfi > STATUS REACTOR * UNDER $SMGR
SPIFFY   REACTOR  PURPLE UNDER $SMGR
SPIFFY   BOILER   ELEMENT1 UNDER $SMGR Down
SPIFFY   BOILER   ELEMENT2 UNDER $SMGR Pending
SPIFFY   BOILER   ELEMENT3 UNDER $SMGR Up
SPIFFY   VALVE    MIX1 UNDER $SMGR Pending
SPIFFY   VALVE    MIX2 UNDER $SMGR Down
SPIFFY   VALVE    MIX3 UNDER $SMGR Pending
SPIFFY   CHAMBER  COMPOUND1 UNDER $SMGR Up
SPIFFY   CHAMBER  COMPOUND2 UNDER $SMGR Down
SPIFFY   CHAMBER  COMPOUND3 UNDER $SMGR Pending
SPIFFY   REACTOR  YELLOW UNDER $SMGR
SPIFFY   BOILER   STUFFX UNDER $SMGR Pending
SPIFFY   BOILER   STUFFY UNDER $SMGR Down
SPIFFY   BOILER   STUFFZ UNDER $SMGR Pending
SPIFFY   VALVE    FORMULAX UNDER $SMGR Down
SPIFFY   VALVE    FORMULAY UNDER $SMGR Up
SPIFFY   VALVE    FORMULAZ UNDER $SMGR Up
SPIFFY   CHAMBER  SECRETX UNDER $SMGR Down
SPIFFY   CHAMBER  SECRETY UNDER $SMGR Pending
SPIFFY   CHAMBER  SECRETZ UNDER $SMGR Pending
SPIFFY   REACTOR  AMBER UNDER $SMGR
SPIFFY   BOILER   INGREDTA UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTB UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTC UNDER $SMGR Pending
SPIFFY   VALVE    XXX UNDER $SMGR Pending
SPIFFY   VALVE    YYY UNDER $SMGR Down
SPIFFY   VALVE    ZZZ UNDER $SMGR Pending
SPIFFY   CHAMBER  AAA UNDER $SMGR Pending
SPIFFY   CHAMBER  BBB UNDER $SMGR Up
SPIFFY   CHAMBER  CCC UNDER $SMGR Pending

DSNM $spfi > STATUS REACTOR * UNDER $SMGR, ONLY
SPIFFY   REACTOR  PURPLE UNDER $SMGR
SPIFFY   REACTOR  YELLOW UNDER $SMGR
SPIFFY   REACTOR  AMBER UNDER $SMGR

DSNM $spfi > STATUS REACTOR * UNDER $SMGR, SUBONLY
SPIFFY   BOILER   ELEMENT1 UNDER $SMGR Down
SPIFFY   BOILER   ELEMENT2 UNDER $SMGR Pending
SPIFFY   BOILER   ELEMENT3 UNDER $SMGR Up
SPIFFY   VALVE    MIX1 UNDER $SMGR Pending
SPIFFY   VALVE    MIX2 UNDER $SMGR Down
SPIFFY   VALVE    MIX3 UNDER $SMGR Pending
SPIFFY   CHAMBER  COMPOUND1 UNDER $SMGR Up
SPIFFY   CHAMBER  COMPOUND2 UNDER $SMGR Down
SPIFFY   CHAMBER  COMPOUND3 UNDER $SMGR Pending
SPIFFY   BOILER   STUFFX UNDER $SMGR Pending
SPIFFY   BOILER   STUFFY UNDER $SMGR Down
SPIFFY   BOILER   STUFFZ UNDER $SMGR Pending
SPIFFY   VALVE    FORMULAX UNDER $SMGR Down
SPIFFY   VALVE    FORMULAY UNDER $SMGR Up
```

```
SPIFFY   VALVE    FORMULAZ UNDER $SMGR Up
SPIFFY   CHAMBER  SECRETX UNDER $SMGR Down
SPIFFY   CHAMBER  SECRETY UNDER $SMGR Pending
SPIFFY   CHAMBER  SECRETZ UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTA UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTB UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTC UNDER $SMGR Pending
SPIFFY   VALVE    XXX UNDER $SMGR Pending
SPIFFY   VALVE    YYY UNDER $SMGR Down
SPIFFY   VALVE    ZZZ UNDER $SMGR Pending
SPIFFY   CHAMBER  AAA UNDER $SMGR Pending
SPIFFY   CHAMBER  BBB UNDER $SMGR Up
SPIFFY   CHAMBER  CCC UNDER $SMGR Pending

DSNM $spfi > STATUS REACTOR * UNDER $SMGR, UP
SPIFFY   BOILER   ELEMENT3 UNDER $SMGR Up
SPIFFY   CHAMBER  COMPOUND1 UNDER $SMGR Up
SPIFFY   VALVE    FORMULAY UNDER $SMGR Up
SPIFFY   VALVE    FORMULAZ UNDER $SMGR Up
SPIFFY   CHAMBER  BBB UNDER $SMGR Up

DSNM $spfi > STATUS REACTOR * UNDER $SMGR, NOT-UP
SPIFFY   BOILER   ELEMENT1 UNDER $SMGR Down
SPIFFY   BOILER   ELEMENT2 UNDER $SMGR Pending
SPIFFY   VALVE    MIX1 UNDER $SMGR Pending
SPIFFY   VALVE    MIX2 UNDER $SMGR Down
SPIFFY   VALVE    MIX3 UNDER $SMGR Pending
SPIFFY   CHAMBER  COMPOUND2 UNDER $SMGR Down
SPIFFY   CHAMBER  COMPOUND3 UNDER $SMGR Pending
SPIFFY   BOILER   STUFFX UNDER $SMGR Pending
SPIFFY   BOILER   STUFFY UNDER $SMGR Down
SPIFFY   BOILER   STUFFZ UNDER $SMGR Pending
SPIFFY   VALVE    FORMULAX UNDER $SMGR Down
SPIFFY   CHAMBER  SECRETX UNDER $SMGR Down
SPIFFY   CHAMBER  SECRETY UNDER $SMGR Pending
SPIFFY   CHAMBER  SECRETZ UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTA UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTB UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTC UNDER $SMGR Pending
SPIFFY   VALVE    XXX UNDER $SMGR Pending
SPIFFY   VALVE    YYY UNDER $SMGR Down
SPIFFY   VALVE    ZZZ UNDER $SMGR Pending
SPIFFY   CHAMBER  AAA UNDER $SMGR Pending
SPIFFY   CHAMBER  CCC UNDER $SMGR Pending

DSNM $spfi > STATUS REACTOR AMBER UNDER $SMGR, NOT-UP
SPIFFY   VALVE   INGREDTA UNDER $SMGR Pending
SPIFFY   VALVE   INGREDTB UNDER $SMGR Pending
SPIFFY   VALVE   INGREDTC UNDER $SMGR Pending
SPIFFY   BOILER  XXX UNDER $SMGR Pending
SPIFFY   BOILER  YYY UNDER $SMGR Down
SPIFFY   BOILER  ZZZ UNDER $SMGR Pending
```

```
SPIFFY   CHAMBER AAA UNDER $SMGR Pending
SPIFFY   CHAMBER CCC UNDER $SMGR Pending

DSNM $spfi > STATUS REACTOR * UNDER $SMGR, DOWN
SPIFFY   BOILER   ELEMENT1 UNDER $SMGR Down
SPIFFY   VALVE    MIX2 UNDER $SMGR Down
SPIFFY   CHAMBER  COMPOUND2 UNDER $SMGR Down
SPIFFY   BOILER   STUFFY UNDER $SMGR Down
SPIFFY   VALVE    FORMULAX UNDER $SMGR Down
SPIFFY   CHAMBER  SECRETX UNDER $SMGR Down
SPIFFY   VALVE    YYY UNDER $SMGR Down

DSNM $spfi > STATUS REACTOR * UNDER $SMGR, NOT-DOWN
SPIFFY   BOILER   ELEMENT2 UNDER $SMGR Pending
SPIFFY   BOILER   ELEMENT3 UNDER $SMGR Up
SPIFFY   VALVE    MIX1 UNDER $SMGR Pending
SPIFFY   VALVE    MIX3 UNDER $SMGR Pending
SPIFFY   CHAMBER  COMPOUND1 UNDER $SMGR Up
SPIFFY   CHAMBER  COMPOUND3 UNDER $SMGR Pending
SPIFFY   BOILER   STUFFX UNDER $SMGR Pending
SPIFFY   BOILER   STUFFZ UNDER $SMGR Pending
SPIFFY   VALVE    FORMULAY UNDER $SMGR Up
SPIFFY   VALVE    FORMULAZ UNDER $SMGR Up
SPIFFY   CHAMBER  SECRETY UNDER $SMGR Pending
SPIFFY   CHAMBER  SECRETZ UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTA UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTB UNDER $SMGR Pending
SPIFFY   BOILER   INGREDTC UNDER $SMGR Pending
SPIFFY   VALVE    XXX UNDER $SMGR Pending
SPIFFY   VALVE    ZZZ UNDER $SMGR Pending
SPIFFY   CHAMBER  AAA UNDER $SMGR Pending
SPIFFY   CHAMBER  BBB UNDER $SMGR Up
SPIFFY   CHAMBER  CCC UNDER $SMGR Pending
```

Sample User-Written Code for SPIFFY Subsystem Interface Process

The following is the user-written SPIFFY subsystem interface code, which, when compiled, is bound in with the Tandem I process program frame code to create the SPIFFY I process object file. This example does not illustrate the processing of any state-change commands, nor does it illustrate how to handle command cancellation (_EV^CANCEL).

Note. Refer to Section A, "DSNM Library Services," for detailed descriptions of the library services that appear in this example.

```
?INSPECT,SYMBOLS,NOCODE,NOMAP,SAVEABEND

?SETTOG 1                                       ! Puts me in test mode

?SOURCE KDSNDEFS ( IPROCESS^DEFINITIONS )

BLOCK PRIVATE; ! My globals

-- Error cache

--  CONSTANT ZDSN-ERR-INTERNAL-ERR           VALUE -34.
--  CONSTANT ZDSN-ERR-OBJTYPE-NOT-SUPPORTED  VALUE -56.
--  CONSTANT ZDSN-ERR-CMD-NOT-SUPP           VALUE -69.
--  CONSTANT ZDSN-ERR-UNSUPPORTED-BY-SUBSYS  VALUE -77.
--  CONSTANT ZDSN-ERR-UNSUPPORTED-BY-I       VALUE -78.
--  CONSTANT ZDSN-ERR-MISSING-OBJTYPE        VALUE -81.
--  CONSTANT ZDSN-ERR-BADOBJTYPE             VALUE -82.

! SPIFFY subsystem definitions

?SOURCE SPIFDEFS

------------------------------------------------------------------------
------------------------------------------------------------------------
--------------   Contents of SPIFDEFS  -------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
! CODES.
LITERAL
  ! Types of objects
  LEAST^LITTLE^THING, REACTOR = LEAST^LITTLE^THING,
  BOILER, VALVE, CHAMBER, ASSEMBLY, COGWHEEL, GEAR, OTHER,
  ALMOST^ANYTHING = OTHER, ANYTHING,

  ! States of objects
  STOPPED, GOING,                          ! ASSEMBLY states
  LOCKED, ROTATING, COASTING, IDLE,        ! COGWHEEL and GEAR
                                           ! states

  ! Operations on objects
  TELLABOUT,                               ! Operations for
                                           ! all objects
  WARMUP, HEATUP, COOLOFF, SHUTOFF,        ! REACTOR component
                                           ! operations
  GO, DONTGO,                              ! ASSEMBLY operations
  LOOSENUP, SPEEDUP, SLOWUP, LOCKUP,       ! COGWHEEL and GEAR
                                           ! operations

  ! Characteristics of objects
  BLACK, YELLOW, CYAN, MAGENTA,            ! ASSEMBLY, COGWHEEL,
                                           ! GEAR in 4 colors
  IRON, STEEL, COPPER, BRASS;              ! made 4 ways
```

```
! ERRORS
LITERAL COMMAND^DONE, COMMAND^MSG^TOOSHORT, COMMAND^INVALID,
        TYPE^INVALID, COMMAND^INVALID^FOR^TYPE,
        COMMAND^IMPOSSIBLE, THING^NONEXISTENT,
        POP^NONEXISTENT;

literal spiffy^name^len= 20, max^response^things = 2;

STRUCT spiffy^thing^things =
   INT type;                          ! object type
   INT name^occurs;                   ! length of object name
   STRING name [0:spiffy^name^len-1];  ! object name
   INT pop^name^occurs;               ! length of name of
                                      ! object's parent
   STRING pop^name[0:spiffy^name^len-1];   ! parent name
   ! For ASSEMBLYs and subordinates
   INT state;                         ! ASSEMBLY state
   INT color;                         ! and other essentials
   INT composition;
   ! For the various REACTOR components
   INT(32) temp;                      ! Degrees Kelvin
   INT(32) press;                     ! MM Hg
   INT(32) vol;                       ! Liters
   INT(32) n;                         ! Number of elements
DEFINE starname = "*           "#;
DEFINE noname = "             "#;

STRUCT spiffy^thing^def (*); BEGIN
    spiffy^thing^things;
END;

STRUCT spiffy^command^def (*); ! Command message, which also
                               ! heads response message
  BEGIN
   INT cmd;
   INT type;
   INT response^context;
   STRING name[0:spiffy^name^len-1];
   STRING pop^name[0:spiffy^name^len-1];
  END;

STRUCT spiffy^response^def (*); ! Response message, which
                                ! contains cmd msg struct
  BEGIN
   STRUCT cmd (spiffy^command^def);
   INT error;
   INT response^things;
   STRUCT thing (spiffy^thing^def) [0:max^response^things-1];
  END;

------------------------------------------------------------------------
------------------------------------------------------------------------
--------------   End of SPIFDEFS  -------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
```

```
! Thread states

LITERAL st^new^object = _ST^MIN^THREAD^STATE, st^prilim^done, st^done,
        st^exec, st^exec^done;

! cx.cf Command flag bit definitions (see cx.def struct) ON        OFF
LITERAL c^info      = _BITDEF(0), ! Command:      Info type    Action type
    c^things      = _BITDEF(1), !                 Things found Not found
    c^fromcmd     = _BITDEF(2), ! Object:         From cmd     From hierarchy
    c^replydetail = _BITDEF(3), ! Reply text:     Detail       Normal
    c^replystate  = _BITDEF(4), ! State in reply: State        No state
    c^errsuppress = _BITDEF(5), ! Error obj:      Suppress     Include
    c^errdetail   = _BITDEF(6), ! Error text:     Detail       Brief
    c^resetstats  = _BITDEF(7), ! Statistics:     Reset        Don't reset
    c^cmdobj      = _BITDEF(8), ! Apply cmd to:   Cmd obj      Not cmd obj
    c^subobj      = _BITDEF(9), !                 Sub obj      Not sub obj
    c^greenstate  = _BITDEF(10),!                 Green obj    Not gr obj
    c^redstate    = _BITDEF(11),!                 Red obj      Not red obj
    c^yellowstate = _BITDEF(12),!                 Yellow obj  Not yellow obj
    c^anystate    = _BITDEF(13),!                 Any state   Colored states
    c^starobj     = _BITDEF(14);! Object name *

! Input and general working list member definition

STRUCT object^lm^def (*);
  BEGIN
    _INPUT^LM^HEADER;      ! generates FOBJ; see Section 3 and Appendix A
    INT cf;                   ! Command flags
    INT dsnmstate;            ! DSNM state of this thing
    INT er;                   ! DSNM or FS Error
    INT spifer;               ! SPIFFY subsystem error
    STRUCT thing (spiffy^thing^def);   ! Particulars about this thing
  END;

! Frame output list member definition

STRUCT frame^output^lm^def (*);
  BEGIN
    _OUTPUT^LM^HEADER;     ! generates FOBJ; see Section 3 and Appendix A
  END;

! Command thread context definition

STRUCT cx^def (*);
  BEGIN
    _COMMAND^CONTEXT^HEADER;
    _CI^ID (spif);
    INT .EXT inobj (object^lm^def);
    INT .EXT currentobj (object^lm^def);
    INT cf;                              ! Command flags
    INT hmodf;                           ! HMOD command flags
    STRUCT r (spiffy^response^def);      ! Command and response area
       STRUCT cmd (spiffy^command^def) = r;
    _LISTPOINTER (current^in);           ! Pointers to working lists
    _LISTPOINTER (current^out);
    _LIST (things);                      ! Working lists
    _LIST (other^things);
  END;
```

```
! SPIFFY Subsystem CI and Subsystem configurations

STRING .spifclass[0:ZDSN^MAX^CICLASS-1] := "SPIFMON ";
STRING .spifsys[0:ZDSN^MAX^SUBSYS-1] := "SPIFFY  ";
INT .EXT spifmon (_CI^DEF);
INT .EXT spiffy  (_SUBSYS^DEF);

END BLOCK;

! Other toolkit necessities

?SOURCE KDSNDEFS ( IPROCESS^GLOBALS )

?NOLIST, SOURCE EXTDECS0(DEBUG,PROCESS_STOP_?,DNUMOUT)

?SOURCE KDSNDEFS ( IPROCESS^EXTDECS )

?LIST

------------------------------------------------------------------------
------------------------------------------------------------------------
--------------   TOOLKIT Required Procs   ----------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------

INT PROC _STARTUP^MODE (component, compiled^in^testmode,
                        accept^startup^component,
                        subject ) EXTENSIBLE;

-- proc returns error code

STRING .EXT component;              -- OUT:OPT  component name, default blank
                                    -- defined by ZDSN^DDL^OBJNAME^DEF
INT .EXT compiled^in^testmode;      -- OUT:OPT  any non-zero = YES, default 0
INT .EXT accept^startup^component;  -- OUT:OPT  any non-zero = YES, default 0
STRING .EXT subject;

BEGIN
  compiled^in^testmode := _COMPILED^IN^TESTMODE;
  accept^startup^component := 0;
  RETURN 0;
END;

INT PROC _STARTUP (cxl, inputl) EXTENSIBLE;

INT .cxl, .inputl;

BEGIN
  STRING errtext[0:29] := ["Invalid SPIFFY configuration",0];
  cxl := $LEN (cx^def);             ! Command thread context length
  inputl := $LEN (object^lm^def);   ! Frame input object list member length

    ! Get CI and subsystem configurations

    IF _ISNULL (@spifmon := _ADD^CI (spifclass))
       OR _ISNULL (@spiffy := _ADD^SUBSYS (spifsys))
    THEN CALL _REPORT^STARTUP^ERROR (0, _EMS^EVENT^FATAL, errtext);

    RETURN ZDSN^ERR^NOERR;

END;
```

```
------------------------------------------------------------------------
------------------------------------------------------------------------
-------------- Command Thread Auxiliary Procs --------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------

INT PROC append^numeric^resulttext (frameobj^arg,num);
INT(32) num;
INT .EXT frameobj^arg;
BEGIN

  ! Use DNUMOUT for conversions because it returns the length of
  ! the converted field (and accepts an extended string pointer
  ! besides).  Return ZDSN^ERR if error occurs, probably memory.

  INT .EXT frameobj (frame^output^lm^def) = frameobj^arg;

  INT len, error := 0;

  STRING text[0:23];

  IF (len := DNUMOUT (text, num, 10)) THEN
      error := _APPEND^OUTPUT (frameobj, ZDSN^VTY^RESULTTEXT,,,text, len);
  RETURN error;
END;

_RC^TYPE PROC format^error^object (obj^arg);
INT .EXT obj^arg;

BEGIN

    ! Generate an error output object for an input object in error.
    ! Er:
    !  < 0 - ZDSN^ERR number; caller must append resulttext if any
    !  > 0 - File system error.  Generate ZDSN^ERR^FS^ERR and put
    !        file system err into resulttext.  Its object is the manager,
    !        which is already in the response object.
    !  = 0 - Shouldn't occur, but treat as ZDSN^ERR^NOERR

    ! If spifer is present with ZDSN^ERR^SUBSYSTEM^ERR then put it
    ! into result text; otherwise ignore it.

    INT .EXT cx (cx^def) = _THREAD^CONTEXT^ADDRESS;
    INT .EXT inobj (object^lm^def) = obj^arg;
    INT .EXT frameobj (frame^output^lm^def);
    INT er;

    IF _ON (inobj.cf, c^errsuppress) THEN
      RETURN _RC^NULL;

    IF _ISNULL(@frameobj := _PUT^LM(cx._OUTPUT.OBJECTLIST,,$LEN(frameobj)))
       THEN RETURN _RC^ABORT (ZDSN^ERR^MEMORY);

    IF er := _FOBJECT^INIT (frameobj.FOBJ, inobj.FOBJ) THEN
       RETURN _RC^ABORT (er);

    IF (frameobj.FOBJ.Z^RESULT := inobj.er) <= 0 THEN
        BEGIN
          IF inobj.er = ZDSN^ERR^SUBSYSTEM^ERR
             AND (er := append^numeric^resulttext (frameobj,
                                                   $DBL (inobj.spifer)))
          THEN RETURN _RC^ABORT (er);

        END
```

```
    ELSE
        BEGIN ! FS Error
          frameobj.FOBJ.Z^RESULT := ZDSN^ERR^FS^ERR;
          IF (er := append^numeric^resulttext (frameobj, $DBL (inobj.er)))
              THEN RETURN _RC^ABORT(er);
        END;

    _RELEASE^OUTPUT (frameobj);

    RETURN _RC^NULL;

END;

_RC^TYPE PROC format^normal^object (obj^arg);

INT .EXT obj^arg;

BEGIN

    ! Generate output object for an input object
    INT .EXT cx (cx^def) = _THREAD^CONTEXT^ADDRESS;
    INT .EXT inobj (object^lm^def) = obj^arg;
    INT .EXT frameobj (frame^output^lm^def);
    INT er;

    IF _OFF (inobj.cf, c^info) THEN RETURN _RC^NULL;

    IF _ISNULL(@frameobj := _PUT^LM (cx._OUTPUT.OBJECTLIST,,$LEN (frameobj)))
        THEN RETURN _RC^ABORT (ZDSN^ERR^MEMORY);

    IF er := _FOBJECT^INIT (frameobj.FOBJ, inobj.FOBJ) THEN
        RETURN _RC^ABORT (er);

    IF _ON (inobj.cf, c^replystate) THEN
        frameobj.FOBJ.Z^RESULT := inobj.dsnmstate

    ELSE frameobj.FOBJ.Z^RESULT := 0;

    _RELEASE^OUTPUT (frameobj);

    RETURN _RC^NULL;

END;
```

Note. This does not illustrate the processing of an INFO command, and shows incomplete processing of the STATUS command. It only produces output for a BRIEF response modifier; it does not produce the output for a DETAIL RMOD.

```
------------------------------------------------------------------------
------------------------------------------------------------------------
-------------- Command Thread _THREAD Procs ----------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------

_THREAD^PROC (info^cmd^proc);   FORWARD;

_THREAD^PROC (action^cmd^proc); FORWARD;

_THREAD^PROC (_COMMAND^PROC);
  BEGIN
    ! First command thread proc.  Analyze command and process objects
    ! one at a time until we are done.  Then stop.

    INT .EXT cx (cx^def) = _THREAD^CONTEXT^ADDRESS;
    INT .EXT inobj (object^lm^def), .EXT outobj (object^lm^def);
    INT .EXT thing (spiffy^thing^def);
    INT er,k;
    _RC^TYPE rc;
    INT smodf, amodf, rmodf, emodf;
    _LISTPOINTER (temp);
```

```
! Return cmd flags for HMOD value
INT SUBPROC hmodflags (xhmod);
INT xhmod;
  BEGIN
    CASE xhmod OF
      BEGIN
        ZDSN^HMOD^ONLY    -> RETURN c^cmdobj;
        ZDSN^HMOD^SUBONLY -> RETURN c^subobj;
        OTHERWISE         -> RETURN c^cmdobj + c^subobj;
      END;
  END;

! Return cmd flags for SMOD value
INT SUBPROC smodflags (xsmod);
INT xsmod;
BEGIN
    CASE xsmod OF BEGIN
    ZDSN^SMOD^RED       -> RETURN c^redstate;
    ZDSN^SMOD^GREEN     -> RETURN c^greenstate;
    ZDSN^SMOD^NOT^RED   -> RETURN c^greenstate + c^yellowstate;
    ZDSN^SMOD^NOT^GREEN -> RETURN c^redstate   + c^yellowstate;
    OTHERWISE           -> RETURN c^redstate   + c^yellowstate +
                                  c^greenstate + c^anystate;
    END;
END;

! Return cmd flags for EMOD value
INT SUBPROC emodflags (xemod);
INT xemod;

BEGIN
    CASE xemod OF
      BEGIN
        ZDSN^EMOD^SUPPRESS  -> RETURN c^errsuppress;
        ZDSN^EMOD^DETAIL    -> RETURN c^errdetail;
        OTHERWISE           -> RETURN 0;
      END;
END;

! Return cmd flags for RMOD value
INT SUBPROC rmodflags (xrmod);
INT xrmod;
BEGIN
    RETURN IF xrmod = ZDSN^RMOD^DETAIL THEN c^replydetail
                                       ELSE 0;
END;

! Return cmd flags for AMOD value
INT SUBPROC amodflags (xamod);
INT xamod;
BEGIN
    RETURN IF xamod = ZDSN^AMOD^RESET THEN c^resetstats
                                      ELSE 0;
END;
```

```
    ! Return SPIFFY type code for type name
    INT SUBPROC typecode (tname);
    STRING .EXT tname;
    BEGIN;
        IF      tname = "REACTOR " THEN RETURN reactor
        ELSE IF tname = "BOILER  " THEN RETURN boiler
        ELSE IF tname = "VALVE   " THEN RETURN valve
        ELSE IF tname = "CHAMBER " THEN RETURN chamber
        ELSE IF tname = "ASSEMBLY" THEN RETURN assembly
        ELSE IF tname = "COGWHEEL" THEN RETURN cogwheel
        ELSE IF tname = "GEAR    " THEN RETURN gear
        ELSE RETURN other;
    END;

    ! Boolean, true if type has subordinates
    INT SUBPROC has^subordinates (type);
    INT type;
    BEGIN
        CASE type OF
          BEGIN
           reactor, assembly, other, anything -> RETURN 1;
           OTHERWISE                          -> RETURN 0;
          END;
    END;

---------- Procedure Body ----------------------

    ! First analyze the command modifiers and the action code to
    ! set command flags that will determine the major paths through
    ! the thread procedure states.

    CASE _THREAD^STATE OF BEGIN
    _ST^INITIAL ->
        ! Flags for modifiers
        cx.hmodf := hmodflags (cx._INPUT.MOD.Z^HMOD);
        smodf    := smodflags (cx._INPUT.MOD.Z^SMOD);
        emodf    := emodflags (cx._INPUT.MOD.Z^EMOD);
        rmodf    := rmodflags (cx._INPUT.MOD.Z^RMOD);
        amodf    := amodflags (cx._INPUT.MOD.Z^AMOD);

        ! Set command flags by action
        CASE cx._INPUT.ACTION OF BEGIN
        ZDSN^ACTION^STATUS  ->
            cx.cf := rmodf + emodf + smodf + c^info + c^replystate;
        ZDSN^ACTION^INFO ->
            cx.cf := rmodf + emodf + c^info;
        ZDSN^ACTION^STATISTICS ->
            cx.cf := rmodf + emodf + amodf + c^info;
        ZDSN^ACTION^START, ZDSN^ACTION^STOP, ZDSN^ACTION^ABORT ->
            cx.cf := emodf + smodf;
        ZDSN^ACTION^AGGREGATE ->
            cx.hmodf := hmodflags(0);
            cx.cf := smodflags(0) + emodflags(0) + cx.hmodf + c^info;
        OTHERWISE ->
            RETURN _RC^ABORT (ZDSN^ERR^CMD^NOT^SUPP);
        END;
        _TURNON (cx.cf, c^fromcmd); ! Everything comes from the command
                                    ! at first

        ! Having analyzed the command and modifiers, process command
        ! objects one at a time.
        ! Redispatch this thread procedure in the new^object state to get
        ! first command object for processing.

        _DISPATCH^THREAD (, st^new^object, _EV^CONTINUE);
```

```
st^new^object ->
    ! Enter this state each time we need a new object from the
    ! frame's input object list, which occurs initially and after
    ! the preceding object has been processed completely.

    ! Get the next object from the _INPUT.OBJECTLIST; when it's _NULL,
    ! we've processed them all.

    if _ISNULL (@cx.inobj := @inobj := _GET^LM (cx._INPUT.OBJECTLIST))
        THEN RETURN _RC^STOP;

    ! As soon as we get the next input object, process its hmod and
    ! set the command flags in the object lm.  We must analyze the hmod
    ! now because it is redefined by the result code.

    inobj.cf := cx.cf + (IF inobj.FOBJ.Z^HMOD
                         THEN hmodflags (inobj.FOBJ.Z^HMOD)
                         ELSE cx.hmodf);
    inobj.FOBJ.Z^RESULT := 0;

    ! Set up the command elements and open the manager

    IF inobj.FOBJ.Z^SUBSYS <> "SPIFFY  " THEN
      BEGIN
        ! Found an error.  Put out an error object and redispatch
        ! this proc in the current state to process the next one.
        ! If format^error^object returns anything but _RC^NULL, it
        ! found an error of its own which supersedes this one to
        ! abort the thread.

        inobj.er := ZDSN^ERR^BADSUBSYS;

        IF (rc := format^error^object (inobj)) <> _RC^NULL THEN
           RETURN rc;

        _DISPATCH^THREAD (,,_EV^CONTINUE);

      END;

    IF (inobj.thing.type := typecode (inobj.FOBJ.Z^OBJTYPE)) = other
      THEN BEGIN
        inobj.er := ZDSN^ERR^OBJTYPE^NOT^SUPPORTED;
        IF (rc := format^error^object (inobj)) <> _RC^NULL THEN
           RETURN rc;
        _DISPATCH^THREAD (,,_EV^CONTINUE);
      END;

    IF NOT has^subordinates (inobj.thing.type) THEN
      BEGIN
        ! If object has no subordinates, don't look for them no
        ! matter what the hmod says.  If further the hmod says not
        ! to process the object, there isn't much to do with it...

        _TURNOFF (inobj.cf,c^subobj);

        IF _OFF (inobj.cf,c^cmdobj) THEN
           _DISPATCH^THREAD (,,_EV^CONTINUE);
      END;

    IF inobj.FOBJ.Z^OBJNAME = starname THEN
      _TURNON (inobj.cf,c^starobj);

    IF (inobj.er := _OPEN^CI (spifmon, cx.spif, inobj.FOBJ.Z^MANAGER))
      THEN BEGIN
        IF (rc := format^error^object (inobj)) <> _RC^NULL THEN
           RETURN rc;
        _DISPATCH^THREAD (,,_EV^CONTINUE);
      END;
```

```
        ! Set the current input and output lists for processing
        ! thread and put the inobj on the input.
        ! Note:  The current^in and current^out list pointers may be
        !        exchanged several times while processing an object.
        !        Reference after this initialization should always be
        !        made through the current pointers rather than to the
        !        things/other^things lists directly.

        @cx.current^in := @cx.things;

        @cx.current^out := @cx.other^things;
        IF _ISNULL (_PUT^LM (cx.current^in,,$LEN (inobj),inobj)) THEN
            RETURN _RC^ABORT (ZDSN^ERR^MEMORY);

        ! Dispatch a proc to process the command.
        ! Note: All commands begin with (or consist entirely of) a
        !       TELLABOUT command with the exception of an action command on
        !       a single object (i.e., no subordinates, no *).

        IF _ON (inobj.cf, c^info) THEN
          BEGIN

            ! Command requires info (TELLABOUT) command only.
            ! Set state where we wish to return to this proc.

            _THREAD^STATE := st^done;
            _SAVE^THREAD^AND^DISPATCH (@info^cmd^proc, st^new^object,
               _EV^STARTUP);

            ! If _save^thread fails, we fall through to here and ...

            RETURN _RC^ABORT (ZDSN^ERR^MEMORY);
          END;

        ! Must be an action command

        IF _ANYOFF (inobj.cf, c^anystate) OR _ON (inobj.cf, c^subobj) THEN
          BEGIN
            ! Command requires info as a preliminary to its execution.
            _THREAD^STATE := st^prilim^done;      ! State to return to this
                                                  ! proc
            _SAVE^THREAD^AND^DISPATCH (@info^cmd^proc, st^new^object,
              _EV^STARTUP);
            RETURN _RC^ABORT (ZDSN^ERR^MEMORY);
          END;

        ! Can perform action directly

        _THREAD^STATE := st^done;          ! State to return to this proc
        _SAVE^THREAD^AND^DISPATCH (@action^cmd^proc, st^new^object,
          _EV^STARTUP);
        RETURN _RC^ABORT (ZDSN^ERR^MEMORY);

    st^prilim^done ->
        ! We've done a preliminary command to produce a list of objects
        ! for the real command.  Swap the current input and output lists
        ! and get on with the main event.

        @temp := @cx.current^in;
        @cx.current^in := @cx.current^out;
        @cx.current^out := @temp;

        _THREAD^STATE := st^done; ! Return state
        _SAVE^THREAD^AND^DISPATCH (@action^cmd^proc, st^new^object,
          _EV^STARTUP);
        RETURN _RC^ABORT (ZDSN^ERR^MEMORY);
```

```
        st^done ->
            ! Whatever objects resulted from executing the command on
            ! cx.inobj are now on the cx.current^out list.  Build the
            ! thread output for return to the frame.

            CALL _CLOSE^CI (cx.spif);

            WHILE _NOTNULL (@outobj := _GET^LM (cx.current^out)) DO BEGIN
                IF outobj.er THEN rc := format^error^object (outobj)
                ELSE rc := format^normal^object (outobj);
                IF rc <> _RC^NULL THEN RETURN rc;
            END;

            ! Continue with next _input object.
            _DISPATCH^THREAD (, st^new^object, _EV^CONTINUE);

        END;

    _END^THREAD^PROC;

    _THREAD^PROC (info^cmd^proc);

      BEGIN

        ! Process info commands, including a command preliminary to an
        ! action command.  Generate output list of all objects resulting
        ! from expanding the input list through hierarchy and * object
        ! names, together with info about the object from the subsystem
        ! and its DSNM state.

        INT .EXT cx (cx^def) = _THREAD^CONTEXT^ADDRESS;
        INT .EXT inobj (object^lm^def), .EXT outobj (object^lm^def);
        STRUCT .tempobj (object^lm^def);
        _LISTPOINTER (temp);
        INT er,k;
        _RC^TYPE rc;

        _RC^TYPE SUBPROC error^object (erno, spiferno) VARIABLE;
        INT erno, spiferno;

        BEGIN
            ! Generate an output object for an input object in error.
            ! @Inobj must point to the object from which the error resulted.
            ! Create an output object and put error info into it.
            ! If everything works, put thread into new^object state and return
            ! _RC^WAIT; otherwise return appropriate return code to abort
            ! the thread.
            ! Note:  We can fiddle with the thread's state in a subproc
            !        which makes handling the return simpler than in a proc
            !        such as format^error^object.
            ! Erno:     DSNM or FS error
            ! Spiferno: Spiffy error number, if present

            IF _ISNULL (@outobj := _PUT^LM (cx.current^out,,$LEN (outobj))) THEN
              RETURN _RC^ABORT (ZDSN^ERR^MEMORY);

            IF er :=  _FOBJECT^INIT (outobj.FOBJ, inobj.FOBJ) THEN
              RETURN _RC^ABORT (er);

            outobj.cf := inobj.cf;
            outobj.thing ':=' inobj.thing FOR $LEN (outobj.thing) BYTES;
            outobj.er := erno;

            IF $PARAM (spiferno) THEN outobj.spifer := spiferno;
            _THREAD^STATE := st^new^object;
            RETURN _RC^WAIT;

        END;
```

```
! Return object type string from the SPIFFY type code
SUBPROC typename (tname, typecode);
STRING .EXT tname;
INT typecode;
BEGIN
    CASE typecode OF
      BEGIN
        reactor   -> tname ':=' "REACTOR ";
        boiler    -> tname ':=' "BOILER  ";
        valve     -> tname ':=' "VALVE   ";
        chamber   -> tname ':=' "CHAMBER ";
        assembly  -> tname ':=' "ASSEMBLY";
        cogwheel  -> tname ':=' "COGWHEEL";
        gear      -> tname ':=' "GEAR    ";
        OTHERWISE -> tname ':=' "OTHER   ";
     END;
END;

! Return DSNM state of a thing from its subsystem info
INT SUBPROC dsnmstate (thing^arg);
INT .EXT thing^arg;

BEGIN
    INT .EXT thing (spiffy^thing^def) = thing^arg;

    CASE thing.state OF
      BEGIN
           --  stopped, going,                    ! ASSEMBLY states
           --  locked, idle, coasting, rotating,  ! COGWHEEL, GEAR states

        stopped, locked, idle   -> RETURN ZDSN^STATE^RED;
        coasting                -> RETURN ZDSN^STATE^YELLOW;
        going, rotating         -> RETURN ZDSN^STATE^GREEN;
        OTHERWISE ->
        CASE thing.type OF
          BEGIN
            reactor -> RETURN ZDSN^STATE^NULL;
            boiler, valve, chamber ->
                ! T   0-297  = red
                !   298-595  = yellow
                !   596-893  = green
                !   894-1191 = yellow
                !  1192-up   = red
                IF thing.temp <  298D or thing.temp >= 1192D THEN
                    RETURN ZDSN^STATE^RED
                ELSE IF thing.temp < 596D or thing.temp >= 894D THEN
                    RETURN ZDSN^STATE^YELLOW
                ELSE
                    RETURN ZDSN^STATE^GREEN;
            OTHERWISE -> RETURN ZDSN^STATE^UNKNOWN;
          END;
    END;
END;

! Return true if xstate satisfies state flags (smodf flags) in
! in xcf; false otherwise.
```

```
INT SUBPROC state^ok (xstate,xcf);
INT xstate, xcf;
BEGIN
  CASE xstate OF
    BEGIN
      ZDSN^STATE^GREEN ->   RETURN _ON (xcf,c^greenstate);
      ZDSN^STATE^YELLOW ->  RETURN _ON (xcf,c^yellowstate);
      ZDSN^STATE^RED ->     RETURN _ON (xcf,c^redstate);
      OTHERWISE ->          RETURN _ON (xcf,c^anystate);
    END;

END;

! Produce a normal (non-error) output object
_RC^TYPE SUBPROC normal^object (thing^arg);

INT .EXT thing^arg;
INT er;

BEGIN
  INT .EXT thing (spiffy^thing^def) = thing^arg;
  INT state;
  state := dsnmstate (thing);
  IF _OFF (inobj.cf, c^starobj) AND NOT state^ok (state, inobj.cf) THEN
      RETURN _RC^NULL;

  IF _ISNULL (@outobj := _PUT^LM (cx.current^out,,$LEN (outobj))) THEN
      RETURN _RC^ABORT (ZDSN^ERR^MEMORY);

  outobj.cf := inobj.cf;
  outobj.thing ':=' thing for $LEN (outobj.thing) BYTES;

  _TURNON (outobj.cf, c^things);

  outobj.dsnmstate := state;

  IF thing.name = inobj.FOBJ.Z^OBJNAME FOR spiffy^name^len BYTES THEN
    BEGIN
      ! Same as command thing.
      IF er := _FOBJECT^INIT (outobj.FOBJ, inobj.FOBJ) THEN
         RETURN _RC^ABORT (er);

      IF _OFF (inobj.cf, c^things) THEN
        BEGIN
          inobj.thing ':=' thing FOR $LEN (inobj.thing) BYTES;
          inobj.dsnmstate := state;
          _TURNON (inobj.cf, c^things);
        END;

    END

  ELSE
    BEGIN
      ! Subordinate of or derived from command thing
      IF er := _FOBJECT^INIT (outobj.FOBJ,, inobj.FOBJ) THEN
         RETURN _RC^ABORT (er);
      outobj.FOBJ.Z^OBJNAME ':=' [ ZDSN^MAX^OBJNAME * [" "]];
      outobj.FOBJ.Z^OBJNAME ':=' thing.name FOR spiffy^name^len BYTES;
      CALL typename (outobj.FOBJ.Z^OBJTYPE,thing.type);

      IF _ON (inobj.cf, c^starobj) THEN _TURNOFF (outobj.cf, c^starobj)

      ELSE _TURNOFF (outobj.cf, c^fromcmd + c^subobj);

    END;

  RETURN _RC^NULL;

END;
```

```
CASE _THREAD^STATE OF BEGIN
st^new^object ->
    IF _ISNULL (@inobj := @cx.currentobj := _GET^LM (cx.current^in))THEN
      BEGIN
        ! Out of input objects; restore caller and continue.
        ! Note:  Calling proc has set the state in which it
        !        desires to return before saving the thread state
        !        and dispatching this proc.
       _RESTORE^THREAD^AND^DISPATCH (_EV^CONTINUE);
        ! If _restore^thread fails, we fall through to here and ...
       RETURN _RC^ABORT (ZDSN^ERR^NOTPUSHED);
      END;

    ! If we already know things about the input object, it's been
    ! asked about earlier (probably from a * object).  If c^cmdobj
    ! is off, we don't want to know things about it unless it's *.
    ! Either way, skip issuing a command for it and proceed
    ! directly to subordinate processing, if any.
    IF _ON (inobj.cf, c^things) OR _ALLOFF (inobj.cf, c^cmdobj +
                                            c^starobj)
    THEN
     BEGIN
         IF _ON (inobj.cf, c^cmdobj)
            AND (rc := normal^object (inobj.thing)) <> _RC^NULL
         THEN RETURN rc;
        _DISPATCH^THREAD (, st^exec^done, _EV^CONTINUE);
     END;

    ! Note:  Star objects.  A star object is replaced on the input
    ! list for this procedure (cx.current^in) by the things to
    ! which it expands.  This is done without regard for modifiers
    ! (hmod, smod), which must be applied to the resulting objects
    ! rather than *.  Fortunately, * can only come from the
    ! _COMMAND^PROC rather than from a later iteration of this one
    ! (that is to say, this proc never produces a * object in its
    ! output list).  _COMMAND^PROC hands out objects one at a
    ! time, so * always appears alone on this proc's input.

    ! Therefore after processing a *, we put the output back onto
    ! the (now empty) input and iterate this procedure again, this
    ! time paying attention to the modifiers.

    cx.cmd.cmd := tellabout;
    cx.cmd.response^context := 0;
    cx.cmd.type := inobj.thing.type;
    cx.cmd.name ':=' inobj.FOBJ.Z^OBJNAME FOR spiffy^name^len BYTES;
    cx.cmd.pop^name ':=' starname;
    IF (er := _SEND^CI (cx.spif, cx.cmd, $LEN (cx.cmd),$LEN (cx.r))) THEN
        RETURN error^object (er);

    _THREAD^STATE := st^exec;

    RETURN _RC^WAIT;       ! We'll get _EV^IODONE when I/O completes

st^exec ->  ! _EV^IODONE dispatched us
    @inobj := @cx.currentobj;
    IF _CI^LASTERROR (cx.spif) THEN
        RETURN error^object (_CI^LASTERROR (cx.spif));

    IF cx.r.error THEN
        RETURN error^object (ZDSN^ERR^SUBSYSTEM^ERR, cx.r.error);
```

```
            FOR k := 0 to cx.r.response^things-1 DO
              BEGIN
                IF (rc := normal^object (cx.r.thing[k])) <> _RC^NULL THEN
                  RETURN rc;
              END;

            IF cx.r.cmd.response^context THEN
              BEGIN
                ! Note:  If _SEND^CI produces an error now, something happened
                ! to the SPIFFY manager after we last talked to it.
                ! Put object with error into output, even though it may
                ! have appeared earlier.
                IF (er := _SEND^CI (cx.spif, cx.cmd, $LEN (cx.cmd), $LEN (cx.r)))
                    THEN RETURN error^object (er);
                RETURN _RC^WAIT;
              END;

            IF _OFF (inobj.cf,c^starobj) THEN
                _DISPATCH^THREAD (, st^exec^done, _EV^CONTINUE); ! Done with this
                                                                 ! obj

            ! Input was a star object.  Input list now empty.  Get next
            ! inobj to free the last inobj got, and to be sure it's
            ! really empty, then exchange the current input and
            ! output, which effectively replaces the star object with
            ! its expansion.  See note in st^new^object state.

            IF _NOTNULL (@inobj := _GET^LM (cx.current^in)) THEN
              BEGIN
                CALL _REPORT^INTERNAL^ERROR (1, _EMS^EVENT^INFO);
                RETURN _RC^ABORT (ZDSN^ERR^INTERNAL^ERR);
              END;

            @temp := @cx.current^in;
            @cx.current^in := @cx.current^out;
            @cx.current^out := @temp;

            _DISPATCH^THREAD (, st^new^object, _EV^CONTINUE);

        st^exec^done ->
            ! If this wasn't the command object, or if subordinates aren't
            ! wanted, enter new^object state to process next input;
            ! otherwise issue command to get subordinates and return to
            ! exec state to put on output.

            @inobj := @cx.currentobj;

            IF _ANYOFF (inobj.cf, c^fromcmd + c^subobj) THEN
                _DISPATCH^THREAD (, st^new^object, _EV^CONTINUE);

            _TURNOFF (inobj.cf,c^subobj);

            cx.cmd.cmd := tellabout;
            cx.cmd.response^context := 0;
            cx.cmd.type := anything;
            cx.cmd.name ':=' starname;
            cx.cmd.pop^name ':=' inobj.FOBJ.Z^OBJNAME FOR spiffy^name^len BYTES;

            IF (er := _SEND^CI (cx.spif, cx.cmd, $LEN(cx.cmd),$LEN(cx.r))) THEN
                RETURN error^object (er);

            _THREAD^STATE := st^exec;
            RETURN _RC^WAIT;
        END;
    _END^THREAD^PROC;
```

```
_THREAD^PROC (action^cmd^proc);
   BEGIN
    --          ...
  _END^THREAD^PROC;

------------------------------------------------------------------------
------------------------------------------------------------------------
-------------- Command Thread Termination Proc -------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------

_THREAD^TERMINATION^PROC (_COMMAND^TERMINATION^PROC);
  BEGIN
     INT .EXT cx (cx^def) = _THREAD^CONTEXT^ADDRESS;
     INT .EXT inobj (object^lm^def);

      ! Free our lists and return leaving the thread's original
      ! termination code (_THREAD^TERMINATION^CODE) unchanged.
      ! Leave freeing the official input and output lists to the frame.

     CALL _DEALLOCATE^LIST (cx.things);
     CALL _DEALLOCATE^LIST (cx.other^things);
     CALL _CLOSE^CI (cx.spif);
     RETURN _RC^NULL;
  _END^THREAD^TERMINATION^PROC;
```

Configuring SPIFFY Into DSNM

Specify a file containing the following as an IN file to NETCOM:

```
add \SYS.dsnm.subsystem-interface-config.spifi.public-name, &
   SPIFFY-INTERFACE
add \SYS.dsnm.subsystem-interface-config.spifi.default-&
   processname, $?SPF
add \SYS.dsnm.subsystem-interface-config.spifi.open-params, &
   NOWAIT-DEPTH 15
add \SYS.dsnm.subsystem.spiffy.rank             , 4
add \SYS.dsnm.subsystem.spiffy.default-objtype , cogwheel
add \SYS.dsnm.subsystem.spiffy.devicetype       , 0
add \SYS.dsnm.subsystem.spiffy.manager          , SPIFMON
add \SYS.dsnm.subsystem.spiffy.subsystem-interface, SPIFI
add \SYS.dsnm.subsystem.spiffy.flags            , &
    *-OBJ-ALLOWED  MGR-REQUIRED
add \SYS.dsnm.subsystem.spiffy.objtype.1 ,subsys
add \SYS.dsnm.subsystem.spiffy.objtype.2 ,reactor      subsys
add \SYS.dsnm.subsystem.spiffy.objtype.3 ,boiler      reactor
add \SYS.dsnm.subsystem.spiffy.objtype.4 ,valve       reactor
add \SYS.dsnm.subsystem.spiffy.objtype.5 ,chamber     reactor
add \SYS.dsnm.subsystem.spiffy.objtype.6 ,assembly     subsys
add \SYS.dsnm.subsystem.spiffy.objtype.7 ,cogwheel   assembly
add \SYS.dsnm.subsystem.spiffy.objtype.8 ,gear       assembly
```

This adds the SPIFFY I process and subsystem configuration records listed below to a DSNMCONF file:

```
> NETCOM
NetCom 1> SET FILE $dsnm.idev.dsnmconf
Current config file: $DSNM.IDEV.DSNMCONF
NetCom 2> INFO

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   DEFAULT-OBJTYPE
  SEQUENCE
  VALUE       COGWHEEL

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   DEVICETYPE
  SEQUENCE
  VALUE       0
```

```
Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   FLAGS
  SEQUENCE
  VALUE       *-OBJ-ALLOWED  MGR-REQUIRED

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   MANAGER
  SEQUENCE
  VALUE       SPIFMON

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   OBJTYPE
  SEQUENCE    1
  VALUE       SUBSYS

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   OBJTYPE
  SEQUENCE    2
  VALUE       REACTOR     SUBSYS

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   OBJTYPE
  SEQUENCE    3
  VALUE       BOILER    REACTOR
```

```
Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   OBJTYPE
  SEQUENCE    4
  VALUE       VALVE    REACTOR

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   OBJTYPE
  SEQUENCE    5
  VALUE       CHAMBER  REACTOR

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   OBJTYPE
  SEQUENCE    6
  VALUE       ASSEMBLY  SUBSYS

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   OBJTYPE
  SEQUENCE    7
  VALUE       COGWHEEL  ASSEMBLY

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   OBJTYPE
  SEQUENCE    8
  VALUE       GEAR    ASSEMBLY
```

```
Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   RANK
  SEQUENCE
  VALUE       4

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM
  COMPONENT   SPIFFY
  PARAMETER   SUBSYSTEM-INTERFACE
  SEQUENCE
  VALUE       SPIFI

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM-INTERFACE-CONFIG
  COMPONENT   SPIFI
  PARAMETER   DEFAULT-PROCESSNAME
  SEQUENCE
  VALUE       $?SPF

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM-INTERFACE-CONFIG
  COMPONENT   SPIFI
  PARAMETER   OPEN-PARAMS
  SEQUENCE
  VALUE       NOWAIT-DEPTH 15

Record
------
  SYSTEM      \SYS
  SUBSYS      DSNM
  CLASS       SUBSYSTEM-INTERFACE-CONFIG
  COMPONENT   SPIFI
  PARAMETER   PUBLIC-NAME
  SEQUENCE
  VALUE       SPIFFY-INTERFACE
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

X

Z

Special Characters

3. I Process Development Process

A. DSNM Library Services

B. DSNM Error Codes

Examples

Figures

Tables