

# HP NonStop S-Series Server Description Manual

## Abstract

This manual describes the principal architectural features of and the instruction set used by the HP NonStop™ S-series servers. It is written for system analysts and others who require a technical understanding of the server internals.

## Product Version

N.A.

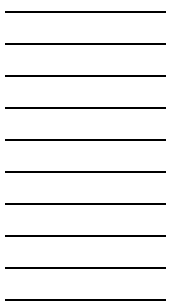
## Supported Release Version Updates (RVUs)

This manual supports G06.21 and all subsequent G-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
520331-003	September 2003

## Document History

Part Number	Product Version	Published
422916-001	N.A.	June 1999
425160-001	N.A.	May 2000
520331-001	N.A.	November 2001
520331-002	N.A.	August 2002
520331-003	N.A.	September 2003



# HP NonStop S-Series Server Description Manual

<a href="#">Glossary</a>	<a href="#">Index</a>	<a href="#">Figures</a>	<a href="#">Tables</a>
--------------------------	-----------------------	-------------------------	------------------------

- [What's New in This Manual](#)   xiii
  - [Manual Information](#)   xiii
  - [New and Changed Information](#)   xiii
- [About This Manual](#)   xv
  - [What's in This Manual](#)   xv
  - [Where to Get More Information](#)   xvi
  - [Notation Conventions](#)   xvii

## 1. Introduction

- [Processor Enclosures and I/O Enclosures](#)   1-2
- [Maximum Server Configuration](#)   1-4
- [Internal Arrangement of System Enclosures](#)   1-6
- [Components of Processor Multifunction \(PMF\) CRU](#)   1-8
- [Fault-Tolerant Process Communication](#)   1-10
- [Fault-Tolerant Disk Access](#)   1-12
- [Maximum Processor Enclosure I/O](#)   1-14
- [Expansion to Second Processor Enclosure](#)   1-16
- [Expansion to External I/O Enclosure](#)   1-18
- [Components of I/O Multifunction \(IOMF\) CRU](#)   1-20
- [Multiple I/O Enclosures](#)   1-22
- [Maximum I/O for a Single Processor Enclosure](#)   1-24
- [Tetrahedral Topology](#)   1-26
  - [First Triangle of Tetrahedron](#)   1-28
- [Tetrahedral Topology With Four Processor Enclosures](#)   1-30
- [Maximum I/O for Four Processor Enclosures](#)   1-32
- [Extending the Tetrahedral Topology](#)   1-34
- [Maximum System With Single Server](#)   1-36
- [ServerNet Clusters](#)   1-36
- [Cluster Topologies for 6770 Switches](#)   1-38
  - [Star Topology](#)   1-40
  - [Split-Star Topology](#)   1-42

<a href="#">Tri-Star Topology</a>	1-44
<a href="#">Layered Topologies for 6780 Switches</a>	1-46
<a href="#">Connections Between Zones in Layered Topology</a>	1-48

## **[2. Principles of System Operation](#)**

<a href="#">ServerNet Devices</a>	2-2
<a href="#">ServerNet Device Identification</a>	2-4
<a href="#">ServerNet Packets</a>	2-6
<a href="#">Packet Transmission and Reception</a>	2-8
<a href="#">Sequence for Outgoing Requests</a>	2-10
<a href="#">Sequence for Responses to Outgoing Requests</a>	2-12
<a href="#">Sequence for Incoming Requests</a>	2-14
<a href="#">Sequence for Responses to Incoming Requests</a>	2-16

## **[3. TNS Data Formats and Number Representations](#)**

<a href="#">TNS Words and RISC Words</a>	3-2
<a href="#">TNS Data Formats</a>	3-4
<a href="#">TNS Byte Instructions</a>	3-6
<a href="#">Instructions for Unsigned and Signed Arithmetic</a>	3-8
<a href="#">Instructions for Decimal and Floating-Point Arithmetic</a>	3-10

## **[4. Memory Addressing and Access](#)**

<a href="#">The Process Address Space</a>	4-2
<a href="#">Organization of the Process Address Space</a>	4-4
<a href="#">Addressing in the Process Address Space</a>	4-6
<a href="#">Address Formats</a>	4-8
<a href="#">Selectable and Flat Logical Segments</a>	4-10
<a href="#">First Four Relative Segments</a>	4-12
<a href="#">Main Stack and SRL Data</a>	4-14
<a href="#">Last Eight Regions</a>	4-16
<a href="#">Native Process Code Allocations</a>	4-18
<a href="#">TNS Process Code Allocations</a>	4-20
<a href="#">Allocation for TNS and Accelerated Code</a>	4-22
<a href="#">Chart of Nonprivileged Space Allocation</a>	4-24
<a href="#">Physical Memory Addressing</a>	4-26
<a href="#">Kseg0 Usage</a>	4-28
<a href="#">Kseg2 Usage</a>	4-30
<a href="#">Kseg1 Memory Access</a>	4-32
<a href="#">Kseg0 Memory Access</a>	4-34
<a href="#">Kseg2 and Nonprivileged Space Memory Access</a>	4-36

<a href="#">The TLBPID Process Identifier</a>	4-38
<a href="#">Nonglobal Address Translation</a>	4-40
<a href="#">Address Translation of Global Elements</a>	4-42
<a href="#">Address-Mapping Tables</a>	4-44
<a href="#">Access of Special Pages</a>	4-46
<a href="#">Defining Unallocated Space</a>	4-48
<a href="#">Context-Bound Addresses</a>	4-50

## **5. Instruction Processing Environments**

<a href="#">Native and TNS Programs and Processes</a>	5-2
<a href="#">Stacks for Native and TNS Processes</a>	5-4
<a href="#">Code and Data Allocations</a>	5-6
<a href="#">Sharing of Code and Data Segments</a>	5-8
<a href="#">Restricting Mode Transitions</a>	5-10

## **6. TNS Execution Modes**

<a href="#">Execution Modes for TNS Compatibility</a>	6-2
<a href="#">TNS Addressing Conventions</a>	6-4
<a href="#">The Environment Register</a>	6-6
<a href="#">The Register Stack</a>	6-8
<a href="#">Register Stack Operations</a>	6-10
<a href="#">The Register Stack in Memory</a>	6-12
<a href="#">Basic P Register Operations</a>	6-14
<a href="#">Branching, Direct and Indirect</a>	6-16
<a href="#">Indexed Addressing in a Code Segment</a>	6-18
<a href="#">Direct and Indirect Addressing in the Data Segment</a>	6-20
<a href="#">Byte Addressing in the Data Segment</a>	6-22
<a href="#">Indexing in the Data Segment</a>	6-24
<a href="#">Examples of Indexing in the Data Segment</a>	6-26
<a href="#">SG Addressing Mode</a>	6-28
<a href="#">Basic Characteristics of Procedures</a>	6-30
<a href="#">Procedure Attributes</a>	6-32
<a href="#">Defining the Procedure's Data</a>	6-34
<a href="#">Data Segment Addressing Modes</a>	6-36
<a href="#">Operations at the Procedure's Top-of-Stack</a>	6-38
<a href="#">Overview of Procedure Call and Exit</a>	6-40
<a href="#">Actions of the PCAL Instruction</a>	6-42
<a href="#">Actions of the EXIT Instruction</a>	6-44
<a href="#">A Procedure's Local Variables</a>	6-46
<a href="#">Passing Parameters to a Called Procedure</a>	6-48

<a href="#">Accessing Parameters in the Called Procedure</a>	6-50
<a href="#">Saving the Stack Frame on a Call</a>	6-52
<a href="#">Restoring a Stack Frame on Return From a Call</a>	6-54
<a href="#">Multiple Markers for Nested Calls</a>	6-56
<a href="#">Returning a Value to the Caller</a>	6-58
<a href="#">Retrieving a Returned Value</a>	6-60
<a href="#">Subprocedure Calls</a>	6-62
<a href="#">Calling External Procedures</a>	6-64
<a href="#">Example of an External Procedure Call</a>	6-66
<a href="#">Resolving Virtual Addresses for External Calls</a>	6-68
<a href="#">An Accelerated Program File in Virtual Memory</a>	6-70
<a href="#">Execution Mode Switches</a>	6-72
<a href="#">Procedure Return in Accelerated Code</a>	6-74
<a href="#">Mapping Return Addresses and Debug Points</a>	6-76
<a href="#">Gateway Tables</a>	6-78
<a href="#">Far Jump Tables</a>	6-80
<a href="#">Maintaining TNS State Values</a>	6-82
<a href="#">Invoking Privilege for CALLABLE Procedures</a>	6-84

## **7. Native Execution Mode**

<a href="#">Native Mode Uses RISC Register Conventions</a>	7-2
<a href="#">RISC Stack Frames</a>	7-4
<a href="#">Procedure Name Spaces for the System Library</a>	7-6
<a href="#">Example of TNS Call to a Native Library Procedure</a>	7-8
<a href="#">Invoking Privilege Requires Taking an Exception</a>	7-10
<a href="#">Stack Switching for Native Privilege Transition</a>	7-12
<a href="#">Example of Enter_Priv Transition</a>	7-14
<a href="#">Far Jumps and Far Gateways Are Needed for SCr</a>	7-16

## **8. Interrupt System**

<a href="#">Interrupt Overview</a>	8-2
<a href="#">Interrupt Sequence</a>	8-4
<a href="#">Interrupt Stack Marker Format</a>	8-6
<a href="#">Transferring Control to an Interrupt Handler</a>	8-8
<a href="#">Interrupt Masking</a>	8-10
<a href="#">TNS Interrupts</a>	8-12

## **9. Interprocessor Communication**

<a href="#">Interprocessor Protocols</a>	9-2
<a href="#">Linker-Listener Protocol</a>	9-4
<a href="#">Message System Protocol</a>	9-6
<a href="#">Message Transfer Mechanisms</a>	9-8
<a href="#">Message Transfer Methods</a>	9-10
<a href="#">Request With Short Request Data</a>	9-12
<a href="#">Request With Medium Request Data</a>	9-14
<a href="#">Request With Long Request Data</a>	9-16
<a href="#">Reply and Reply Acknowledge</a>	9-18

## **10. Input/Output Operations**

<a href="#">Storage and Communications I/O Compared</a>	10-2
<a href="#">Overview of the I/O System</a>	10-4
<a href="#">Layers of I/O Components</a>	10-6
<a href="#">I/O Process Models for Storage I/O</a>	10-8
<a href="#">Storage Operation Queuing</a>	10-10
<a href="#">Packaging and Delivery of Storage Commands</a>	10-12
<a href="#">Storage Read Request Processing</a>	10-14
<a href="#">Storage Write Request Processing</a>	10-16
<a href="#">Communications Operation Queuing</a>	10-18
<a href="#">Application of Communications Queues</a>	10-20
<a href="#">Typical Use of Queue Pairs</a>	10-22
<a href="#">Actions for Empty or Full Queues</a>	10-24
<a href="#">Communications Request Processing</a>	10-26

## **11. TNS Instruction Set**

<a href="#">Memory Addressing Instructions</a>	11-2
<a href="#">Immediate Operand and Shift Instructions</a>	11-4
<a href="#">Boolean Instructions Operate on Stack Registers</a>	11-6
<a href="#">Move, Compare, and Scan Instructions</a>	11-8
<a href="#">Definitions of TNS Instructions</a>	11-10
<a href="#">Additional Operating-System-Only Instructions</a>	11-41
<a href="#">Resource Management</a>	11-42
<a href="#">Memory Management</a>	11-42
<a href="#">List Management</a>	11-42
<a href="#">Tracing</a>	11-42

## [A. TNS Instruction Lists](#)

## [B. TNS Instruction Binary Coding](#)

## [C. TNS Instruction Set Definition](#)

[Symbol Definitions](#) C-1

[Instruction Definitions](#) C-10

[Compatibility Notes](#) C-49

## [Glossary](#)

## [Index](#)

## [Figures](#)

- [Figure 1-1.](#) [A Four-Processor Server Shown Without and With I/O Enclosures](#) 1-3
- [Figure 1-2.](#) [A Maximum Server Would Have 44 System Enclosures](#) 1-5
- [Figure 1-3.](#) [Processor Enclosure Shown From Both Sides](#) 1-7
- [Figure 1-4.](#) [Processor Multifunction \(PMF\) CRU Has Four ServerNet Addressable Controllers](#) 1-9
- [Figure 1-5.](#) [Processes in Different Processors Have Alternative Communication Paths](#) 1-11
- [Figure 1-6.](#) [In Any Processor Enclosure, Both Processors Can Access All Disks](#) 1-13
- [Figure 1-7.](#) [Alternate Configuration of MFIOBs for NonStop S7000, S7400, S70000, and S72000 Servers](#) 1-14
- [Figure 1-8.](#) [Dual MFIOBs Provide Large I/O Capability for a Processor Enclosure](#) 1-15
- [Figure 1-9.](#) [Processor Enclosures Are Interconnected Through ServerNet Expansion Boards](#) 1-17
- [Figure 1-10.](#) [ServerNet Expansion Boards Also Can Connect to I/O Enclosures](#) 1-19
- [Figure 1-11.](#) [I/O Multifunction \(IOMF\) CRU Includes External Enclosure Interface](#) 1-21
- [Figure 1-12.](#) [Any Processor Has Dual Access Paths to All I/O, External or Internal](#) 1-23
- [Figure 1-13.](#) [Dual SEBs Permit Connection of More I/O Enclosures](#) 1-25
- [Figure 1-14.](#) [Tetrahedral Topology Efficiently Connects Processor Enclosures](#) 1-27
- [Figure 1-15.](#) [Tetra 16 Configuration Extends Server Expansion](#) 1-29
- [Figure 1-16.](#) [Adding Fourth Processor Enclosure Completes the Core Tetrahedron](#) 1-31
- [Figure 1-17.](#) [With Dual SEBs, Core Tetrahedron Supports 20 I/O Enclosures](#) 1-33
- [Figure 1-18.](#) [Processor Enclosures Exceeding Four Are Added to Corners](#) 1-35



<a href="#">Figure 1-19.</a>	<a href="#">Eight Processor Enclosures Can Support 36 I/O Enclosures</a>	1-37
<a href="#">Figure 1-20.</a>	<a href="#">ServerNet Cluster Star Topologies</a>	1-39
<a href="#">Figure 1-21.</a>	<a href="#">Eight-Node ServerNet Cluster Using Star Topology</a>	1-41
<a href="#">Figure 1-22.</a>	<a href="#">16-Node ServerNet Cluster Using Split-Star Topology</a>	1-43
<a href="#">Figure 1-23.</a>	<a href="#">24-Node ServerNet Cluster Using Tri-Star Topology</a>	1-45
<a href="#">Figure 1-24.</a>	<a href="#">Single-Zone Cluster Using Layered Topology</a>	1-47
<a href="#">Figure 1-25.</a>	<a href="#">Multiple Zones of ServerNet Clusters Using Layered Topology</a>	1-49
<a href="#">Figure 2-1.</a>	<a href="#">A ServerNet Device Can Be a Processor, Controller, or Bus Interface</a>	2-3
<a href="#">Figure 2-2.</a>	<a href="#">ServerNet ID Identifies ServerNet Devices and Subdevices</a>	2-5
<a href="#">Figure 2-3.</a>	<a href="#">A ServerNet Transaction Has Both a Request and a Response Packet</a>	2-7
<a href="#">Figure 2-4.</a>	<a href="#">BTE Handles Local Requests; AVT Handles Incoming Requests</a>	2-9
<a href="#">Figure 2-5.</a>	<a href="#">Transfer Initiation Logic Uses BTE Descriptors for Outgoing Requests</a>	2-11
<a href="#">Figure 2-6.</a>	<a href="#">BTE Hardware Handles Responses to Outgoing Requests</a>	2-13
<a href="#">Figure 2-7.</a>	<a href="#">Remote Access Logic Uses AVT Table for Managing Incoming Requests</a>	2-15
<a href="#">Figure 2-8.</a>	<a href="#">AVT Hardware Handles Responses to Incoming Requests</a>	2-17
<a href="#">Figure 3-1.</a>	<a href="#">Two TNS Words Are Equivalent to One RISC Word</a>	3-2
<a href="#">Figure 3-2.</a>	<a href="#">A Variety of TNS Data Formats Are Available</a>	3-5
<a href="#">Figure 3-3.</a>	<a href="#">Byte-Manipulating Instructions Assume Big-Endian Numbering</a>	3-7
<a href="#">Figure 3-4.</a>	<a href="#">Formats for TNS Floating-Point and Extended Floating-Point Data</a>	3-11
<a href="#">Figure 4-1.</a>	<a href="#">Nonprivileged and Privileged Spaces Include Global and Nonglobal Areas</a>	4-3
<a href="#">Figure 4-2.</a>	<a href="#">Process Address Space Includes Regions, Unitary Segments, and Pages</a>	4-5
<a href="#">Figure 4-3.</a>	<a href="#">Four Distinct Addressing Areas Exist in the Process Address Space</a>	4-7
<a href="#">Figure 4-4.</a>	<a href="#">Address Formats Are Slightly Different for Each Addressing Area</a>	4-9
<a href="#">Figure 4-5.</a>	<a href="#">Selectable and Flat Logical Segments Differ in Allocation Method</a>	4-11
<a href="#">Figure 4-6.</a>	<a href="#">For TNS Compatibility, the First Four Relative Segments Are Special</a>	4-13
<a href="#">Figure 4-7.</a>	<a href="#">Main Stack and SRL Data for a Process Are in Nonprivileged Space</a>	4-15
<a href="#">Figure 4-8.</a>	<a href="#">The Last Eight Regions Are for Code Addressing</a>	4-17
<a href="#">Figure 4-9.</a>	<a href="#">Native Process Has Two Regions for User Code, Four for Libraries</a>	4-19
<a href="#">Figure 4-10.</a>	<a href="#">TNS Process Has One Region for User Code, One for User Library</a>	4-21

<a href="#">Figure 4-11.</a>	<a href="#">Both TNS and Accelerated Code Are Included in a Code Region</a>	4-23
<a href="#">Figure 4-12.</a>	<a href="#">Kseg0 and Kseg1 Both Map Permanently to Physical Memory</a>	4-27
<a href="#">Figure 4-13.</a>	<a href="#">Kseg0 Is Used Primarily for Addressing System Code and Data</a>	4-29
<a href="#">Figure 4-14.</a>	<a href="#">Kseg2 Provides Absolute Memory Allocations for Every Process</a>	4-31
<a href="#">Figure 4-15.</a>	<a href="#">Access Through Kseg1 Is Direct: No Caching, No Address Translation</a>	4-33
<a href="#">Figure 4-16.</a>	<a href="#">Memory Access Through Kseg0 Uses Memory Caches</a>	4-35
<a href="#">Figure 4-17.</a>	<a href="#">Memory Access Through Nonprivileged Space or Kseg2 Requires Address Translation</a>	4-37
<a href="#">Figure 4-18.</a>	<a href="#">The TLBPID Distinguishes the Address Space of a Process</a>	4-39
<a href="#">Figure 4-19.</a>	<a href="#">Nonglobal Address Translations Require Matching TLBPID</a>	4-41
<a href="#">Figure 4-20.</a>	<a href="#">Address Translation of Global Elements Ignores TLBPID</a>	4-43
<a href="#">Figure 4-21.</a>	<a href="#">On TLB Miss, Nonprivileged Space and Kseg2 Space Translation Uses Address-Mapping Tables</a>	4-45
<a href="#">Figure 4-22.</a>	<a href="#">Special Pages Are Accessed Through Fixed Addresses</a>	4-47
<a href="#">Figure 4-23.</a>	<a href="#">Unallocated Space Is Defined With a Few Null Tables</a>	4-49
<a href="#">Figure 4-24.</a>	<a href="#">Context-Bound Addresses Substitute for Aliases in Unaliased Segments</a>	4-51
<a href="#">Figure 5-1.</a>	<a href="#">Comparing Native and TNS Processes</a>	5-3
<a href="#">Figure 5-2.</a>	<a href="#">A Native Process Uses Two Stacks; TNS Uses Three</a>	5-5
<a href="#">Figure 5-3.</a>	<a href="#">Code and Data Allocations Are Separate and Treated Differently</a>	5-7
<a href="#">Figure 5-4.</a>	<a href="#">Code Segments and Some Data Segments Can Be Shared</a>	5-9
<a href="#">Figure 5-5.</a>	<a href="#">Only Two Kinds of Mode Transitions Are Permitted</a>	5-11
<a href="#">Figure 6-1.</a>	<a href="#">Two Execution Modes Provide TNS Compatibility</a>	6-3
<a href="#">Figure 6-2.</a>	<a href="#">TNS and Accelerated Modes Use TNS Addressing Conventions</a>	6-5
<a href="#">Figure 6-3.</a>	<a href="#">The Environment Register Maintains Procedure State</a>	6-7
<a href="#">Figure 6-4.</a>	<a href="#">The Register Stack Accumulates Arithmetic Results</a>	6-8
<a href="#">Figure 6-5.</a>	<a href="#">The Top-of-Stack Can Occupy Various Positions in the Register Stack</a>	6-9
<a href="#">Figure 6-6.</a>	<a href="#">An Example of Register Stack Operations</a>	6-11
<a href="#">Figure 6-7.</a>	<a href="#">The TNS Register Stack Is Implemented in the RP Wrap Page</a>	6-13
<a href="#">Figure 6-8.</a>	<a href="#">In Concept, the P Register Controls Execution Flow</a>	6-15
<a href="#">Figure 6-9.</a>	<a href="#">The BUN Instruction Is Typical of Branch Instructions</a>	6-16
<a href="#">Figure 6-10.</a>	<a href="#">Two Examples of Branching</a>	6-17
<a href="#">Figure 6-11.</a>	<a href="#">An Example of Code Segment Indexing</a>	6-19
<a href="#">Figure 6-12.</a>	<a href="#">An Example of Indirect Addressing in the Data Segment</a>	6-21
<a href="#">Figure 6-13.</a>	<a href="#">Byte Addressing in the Data Segment Is Restricted to Half Segment</a>	6-22
<a href="#">Figure 6-14.</a>	<a href="#">Indirect Byte Addressing in the Data Segment Can Specify Odd Bytes</a>	6-23

<a href="#">Figure 6-15.</a>	<a href="#">Sequence and Encoding for Indexing in the Data Segment</a>	6-25
<a href="#">Figure 6-16.</a>	<a href="#">Three Examples of Indexing in the Data Segment</a>	6-27
<a href="#">Figure 6-17.</a>	<a href="#">Direct and Indirect Examples of SG Addressing</a>	6-29
<a href="#">Figure 6-18.</a>	<a href="#">Layout of Procedure Code in a TNS Code Segment</a>	6-31
<a href="#">Figure 6-19.</a>	<a href="#">Procedure Entry Points Are Grouped by Attribute in the PEP Table</a>	6-33
<a href="#">Figure 6-20.</a>	<a href="#">Procedure Data Consists of Global Areas and Stack Frame</a>	6-35
<a href="#">Figure 6-21.</a>	<a href="#">There Are Four Data Segment Addressing Modes</a>	6-37
<a href="#">Figure 6-22.</a>	<a href="#">PUSH and POP Instructions Add and Delete Stack Elements</a>	6-39
<a href="#">Figure 6-23.</a>	<a href="#">Example of a Procedure Call and Exit</a>	6-41
<a href="#">Figure 6-24.</a>	<a href="#">Sequence of Events Performed by a PCAL Instruction</a>	6-43
<a href="#">Figure 6-25.</a>	<a href="#">Sequence of Events Performed by an EXIT Instruction</a>	6-45
<a href="#">Figure 6-26.</a>	<a href="#">Defining and Accessing a Procedure's Local Variables</a>	6-47
<a href="#">Figure 6-27.</a>	<a href="#">Example of Passing Parameters to a Called Procedure</a>	6-49
<a href="#">Figure 6-28.</a>	<a href="#">Examples of Accessing Value Parameters and Reference Parameters</a>	6-51
<a href="#">Figure 6-29.</a>	<a href="#">Sequence of Events for Saving a Stack Frame</a>	6-53
<a href="#">Figure 6-30.</a>	<a href="#">Sequence of Events for Restoring a Stack Frame</a>	6-55
<a href="#">Figure 6-31.</a>	<a href="#">Nested Calls Create a Chain of Markers on the Stack</a>	6-57
<a href="#">Figure 6-32.</a>	<a href="#">Example of Returning a Value to the Caller of a Procedure</a>	6-59
<a href="#">Figure 6-33.</a>	<a href="#">Example of Retrieving a Value Returned by a Called Procedure</a>	6-61
<a href="#">Figure 6-34.</a>	<a href="#">Subprocedures Return Values Through Sublocal Data Area</a>	6-63
<a href="#">Figure 6-35.</a>	<a href="#">Saved Copy of the Environment Register Preserves Space ID Index</a>	6-65
<a href="#">Figure 6-36.</a>	<a href="#">Sequence of Events for an External Procedure Call</a>	6-67
<a href="#">Figure 6-37.</a>	<a href="#">Example of Resolving a Virtual Address in User Code Space</a>	6-69
<a href="#">Figure 6-38.</a>	<a href="#">Accelerated Programs Files Contain Both TNS and Accelerated Code</a>	6-71
<a href="#">Figure 6-39.</a>	<a href="#">Switching Modes for System Calls and Translation Assistance</a>	6-73
<a href="#">Figure 6-40.</a>	<a href="#">Accelerated Procedure Return Requires Access to TNS Information</a>	6-75
<a href="#">Figure 6-41.</a>	<a href="#">Pmap Is a Cross-Reference of Code Segment Addresses</a>	6-77
<a href="#">Figure 6-42.</a>	<a href="#">Gateway Tables Provide Privileged-Mode Transitions for Accelerated Code</a>	6-79
<a href="#">Figure 6-43.</a>	<a href="#">Far Jump Tables Are Needed for Calls To and From System Code</a>	6-81
<a href="#">Figure 6-44.</a>	<a href="#">Most TNS State Values Are Kept in RISC General-Purpose Registers</a>	6-83
<a href="#">Figure 6-45.</a>	<a href="#">Invoking Privilege Always Requires Taking an Exception</a>	6-85
<a href="#">Figure 7-1.</a>	<a href="#">RISC Registers Point to Global Data, Main Stack Tip, and Return Address</a>	7-3

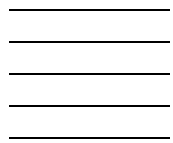
<a href="#">Figure 7-2.</a>	<a href="#">Stack Frames Overlap and Grow to Lower Addresses</a>	7-5
<a href="#">Figure 7-3.</a>	<a href="#">Gateways and To-RISC Shells Enable TNS Access to System Library</a>	7-7
<a href="#">Figure 7-4.</a>	<a href="#">A TNS Call to the System Library Is Directed Through the Shell Map</a>	7-9
<a href="#">Figure 7-5.</a>	<a href="#">syscall Exception or Address Error Exception Sets Privileged State</a>	7-11
<a href="#">Figure 7-6.</a>	<a href="#">Millicode Routines Switch Between Main and Privileged Stacks</a>	7-13
<a href="#">Figure 7-7.</a>	<a href="#">Deliberate Invocation of Error Exception Triggers Privilege Transition</a>	7-15
<a href="#">Figure 7-8.</a>	<a href="#">Far Jump Tables Allow Calls to Cross Direct Jump Area Boundaries</a>	7-17
<a href="#">Figure 8-1.</a>	<a href="#">An Interrupt Can Interrupt a Process or Another Interrupt Handler</a>	8-3
<a href="#">Figure 8-2.</a>	<a href="#">Interrupts Invoke Interrupt Handlers, Return to Interrupted Code</a>	8-5
<a href="#">Figure 8-3.</a>	<a href="#">The Interrupt Stack Marker Saves RISC and TNS State</a>	8-7
<a href="#">Figure 8-4.</a>	<a href="#">The System Interrupt Vector Transfers Control to Software</a>	8-9
<a href="#">Figure 8-5.</a>	<a href="#">Some Interrupts Can Be Masked by Mask Register Bits</a>	8-11
<a href="#">Figure 9-1.</a>	<a href="#">Interprocessor Communication Involves Multiple Levels of Protocol</a>	9-3
<a href="#">Figure 9-2.</a>	<a href="#">From Linker-Listener Perspective: One Request and One Reply</a>	9-5
<a href="#">Figure 9-3.</a>	<a href="#">The Message System Uses Setup and Acknowledgments in Messages</a>	9-7
<a href="#">Figure 9-4.</a>	<a href="#">Message Data Is Transferred To and From Request and Reply Buffers</a>	9-9
<a href="#">Figure 9-5.</a>	<a href="#">Message Transfers Use Three Phases and Two Transfer Methods</a>	9-11
<a href="#">Figure 9-6.</a>	<a href="#">For Request With Short Data, Listener Pre-Pushes Data</a>	9-13
<a href="#">Figure 9-7.</a>	<a href="#">For Request With Medium Data, Listener Post-Pulls Data to Cache</a>	9-15
<a href="#">Figure 9-8.</a>	<a href="#">For Request With Long Data, Listener Post-Pulls Data to Its Buffer</a>	9-17
<a href="#">Figure 9-9.</a>	<a href="#">For All Read Requests, Listener Pre-Pushes Data in Reply Phase</a>	9-19
<a href="#">Figure 10-1.</a>	<a href="#">Application Initiates Storage I/O, Communications I/O Uses Ethernet or ATM</a>	10-3
<a href="#">Figure 10-2.</a>	<a href="#">ServerNet Hardware Bridges Controllers to I/O Software</a>	10-5
<a href="#">Figure 10-3.</a>	<a href="#">Queues in Controllers for Storage, in Processor for Communications</a>	10-7
<a href="#">Figure 10-4.</a>	<a href="#">Alternative Storage I/O Models Provide Backward Compatibility</a>	10-9
<a href="#">Figure 10-5.</a>	<a href="#">Module Driver Pushes Entries to Global Buffers</a>	10-11
<a href="#">Figure 10-6.</a>	<a href="#">I/O Process Sends Command Descriptor Block to Controller</a>	10-13

<a href="#">Figure 10-7.</a>	<a href="#">For Read Request, Controller Pushes Data in Write Transaction</a>	10-15
<a href="#">Figure 10-8.</a>	<a href="#">For Write Request, Controller Pulls Data in Read Transaction</a>	10-17
<a href="#">Figure 10-9.</a>	<a href="#">Communications Controllers Use Work Queues That Are in Processor Memory</a>	10-19
<a href="#">Figure 10-10.</a>	<a href="#">Communications Queues Are Unidirectional and Paired</a>	10-21
<a href="#">Figure 10-11.</a>	<a href="#">Monitor Writes in Outbound Queues, Module Driver Writes in Inbound Queues</a>	10-23
<a href="#">Figure 10-12.</a>	<a href="#">Empty Queue Requires Wakeup Prod, Full Queue Requires Full Notice</a>	10-25
<a href="#">Figure 10-13.</a>	<a href="#">Controller Originates All Work in Most Communications Transfers</a>	10-27
<a href="#">Figure 11-1.</a>	<a href="#">Memory Addressing Instructions Provide Access to the Data Segment</a>	11-2
<a href="#">Figure 11-2.</a>	<a href="#">Examples of Doubleword Addressing</a>	11-3
<a href="#">Figure 11-3.</a>	<a href="#">Examples of Immediate Operand Instructions</a>	11-4
<a href="#">Figure 11-4.</a>	<a href="#">Examples of Logical and Arithmetic Shifts</a>	11-5
<a href="#">Figure 11-5.</a>	<a href="#">Examples of Boolean Instruction Operations</a>	11-6
<a href="#">Figure 11-6.</a>	<a href="#">Examples of Boolean Immediate Instruction Operations</a>	11-7
<a href="#">Figure 11-7.</a>	<a href="#">Moves Can Be Ascending or Descending</a>	11-9
<a href="#">Figure 11-8.</a>	<a href="#">Two Examples of Branch Forward Indirect (BFI)</a>	11-12
<a href="#">Figure 11-9.</a>	<a href="#">Example of Depositing a Field Using the DPF Instruction</a>	11-22
<a href="#">Figure 11-10.</a>	<a href="#">Two Examples of Load Byte From Program (LBP)</a>	11-27
<a href="#">Figure 11-11.</a>	<a href="#">Example of PUSH and POP Instructions</a>	11-35

## Tables

<a href="#">Table 1.</a>	<a href="#">Summary of Contents</a>	xv
<a href="#">Table 3-1.</a>	<a href="#">Ranges of Numbers Representable by Different Data Lengths</a>	3-8
<a href="#">Table 3-2.</a>	<a href="#">Termination Codes for Floating-Point Arithmetic Errors</a>	3-12
<a href="#">Table 4-1.</a>	<a href="#">Summary of Nonprivileged Space Allocation</a>	4-25
<a href="#">Table 8-1.</a>	<a href="#">System Interrupt Vector Entries</a>	8-14
<a href="#">Table A-1.</a>	<a href="#">Alphabetic List of Instructions</a>	A-1
<a href="#">Table A-2.</a>	<a href="#">Categorized List of Instructions</a>	A-9
<a href="#">Table B-1.</a>	<a href="#">Binary Coding, Memory Reference Instructions</a>	B-1
<a href="#">Table B-2.</a>	<a href="#">Binary Coding, Immediate Instructions</a>	B-2
<a href="#">Table B-3.</a>	<a href="#">Binary Coding, Move/Shift/Call/Extended Instructions</a>	B-3
<a href="#">Table B-4.</a>	<a href="#">Binary Coding, Branch Instructions</a>	B-4
<a href="#">Table B-5.</a>	<a href="#">Binary Coding, Stack Instructions</a>	B-5
<a href="#">Table B-6.</a>	<a href="#">Binary Coding, Decimal Arithmetic Instructions</a>	B-7
<a href="#">Table B-7.</a>	<a href="#">Binary Coding, Floating-Point Instructions</a>	B-8
<a href="#">Table C-1.</a>	<a href="#">Definitions of Symbols</a>	C-1

<a href="#">Table C-2.</a>	<a href="#">Instruction Definitions</a>	C-11
----------------------------	---	------



# What's New in This Manual

## Manual Information

### Abstract

This manual describes the principal architectural features of and the instruction set used by the HP NonStop™ S-series servers. It is written for system analysts and others who require a technical understanding of the server internals.

### Product Version

N.A.

### Supported Release Version Updates (RVUs)

This manual supports G06.21 and all subsequent G-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
520331-003	September 2003

### Document History

Part Number	Product Version	Published
422916-001	N.A.	June 1999
425160-001	N.A.	May 2000
520331-001	N.A.	November 2001
520331-002	N.A.	August 2002
520331-003	N.A.	September 2003

## New and Changed Information

The manual has been updated with architectural information to describe the server clustering capabilities that are provided by the HP NonStop ServerNet Switch (model 6780). These switches introduce the concept of the layered topology, which expands the number of supported ServerNet nodes to 64. This topology is described primarily on pages 1-46 through 1-49.

Section 1, Introduction, has also been updated to more accurately reflect actual supported configurations.

This publication has been updated to reflect new product names. Since product names are changing over time, this publication might contain both HP and Compaq product names.





---

---

---

---

# About This Manual

This manual describes the principal architectural features of and the instruction set used by the NonStop S-series servers. It is written for system analysts and others who require a technical understanding of the server internals.

This manual describes information for NonStop S-series servers on G06.21 and subsequent G-series release version updates.

---

**Note.** “NonStop S-series” refers to the hardware that makes up the server. “G-series” refers to the software that runs on the server.

The term “NonStop Sxx000” represents the NonStop S70000, NonStop S72000, NonStop S74000, NonStop S76000, and NonStop S86000 servers.

The term “NonStop S7x00” represents the NonStop S7400 and higher numbered servers. Thus it does not include the NonStop S7000, which is usually designated separately.

---

## What’s in This Manual

[Table 1](#) summarizes the contents of this manual.

---

**Table 1. Summary of Contents** (page 1 of 2)

Section	Title	This section . . .
1	Introduction	Introduces the fundamental architecture of the NonStop S-series servers, starting with the arrangement of system enclosures and then describing progressively larger system topologies.
2	Principles of System Operation	Describes the fundamental operations that drive the architecture described in Section 1. Defines ServerNet devices and ServerNet transactions.
3	TNS Data Formats and Number Representations	Provides a description of some of the data formats used, particularly those that are unique to TNS systems.
4	Memory Addressing and Access	Describes how memory is addressed and accessed within the processors of the NonStop S-series servers.
5	Instruction Processing Environments	Describes the differences between TNS/R native and TNS processes, their stacks, and mode transitions.
6	TNS Execution Modes	Describes the TNS compatibility modes for execution of processes.
7	Native Execution Mode	Describes the basic conventions for native-mode execution of processes.

---

**Table 1. Summary of Contents** (page 2 of 2)

Section	Title	This section . . .
8	Interrupt System	Describes those interrupts that are handled by the NonStop Kernel operating system.
9	Interprocessor Communication	Describes the protocols and transfer mechanisms used by processors to exchange messages through the ServerNet hardware.
10	Input/Output Operations	Describes the input/output architecture for doing both storage I/O and communications I/O through the ServerNet hardware.
11	TNS Instruction Set	Provides text descriptions of the heritage CISC instructions used in TNS systems.
A	TNS Instruction Lists	Lists all TNS instructions with their mnemonics and opcodes.
B	TNS Instruction Binary Coding	Provides several tables that define the binary coding for most TNS instructions.
C	TNS Instruction Set Definition	Provides symbolic definitions of the TNS instruction set.

This manual also contains a glossary of technical terms and abbreviations used throughout the text.

## Where to Get More Information

Manuals describing the NonStop S-series servers are divided into six sets:

Manual Set	Purpose
NonStop S-series server manual set	Provides references for installing, configuring, operating, managing, and supporting NonStop S-series servers.
NonStop S-series subsystem manual set	Provides references for installing, configuring, operating, managing, and supporting NonStop S-series subsystems.
NonStop S-series network manual set	Provides references for installing, configuring, operating, managing, and supporting NonStop S-series networks.

**Manual Set**

NonStop S-series adapter manual set

NonStop S-series configuration and management manual set

OSM and TSM manuals and online help

**Purpose**

Provides references for installing, configuring, operating, managing, and supporting NonStop S-series adapters.

Describes how to configure:

- Entire systems
- Individual hardware and software components such as peripheral devices and communications software

Provides information about using the OSM and TSM packages to bring up and maintain NonStop S-series servers.

This manual is part of the NonStop S-series server manual set.

For more specific details and implementations of the concepts described in this manual, you might want to refer to the *NonStop S-Series Planning and Configuration Guide*. That guide describes how to plan and configure a NonStop S-series server and provides a case study documenting a sample system. It also describes the ServerNet fabrics, the available hardware and software configurations for NonStop S-series servers, site planning and preparation, creating the operational environment, and making hardware and software configuration changes to an existing server. The *NonStop S-Series Planning and Configuration Guide* is written for those who are responsible for planning the installation, configuration, and maintenance of the server and the software environment at a particular site.

In addition, the *NonStop S-Series Planning and Configuration Guide* provides lists and descriptions of all the manuals in each manual set.

## Notation Conventions

### 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 Introduction

This section introduces the fundamental architecture of the NonStop S-Series Sxx000 and S7x00 servers, beginning with a physical overview and then proceeding to describe the components and how they are linked together.

The topics covered in this section are:

[Processor Enclosures and I/O Enclosures](#)

[Maximum Server Configuration](#)

[Internal Arrangement of System Enclosures](#)

[Components of Processor Multifunction \(PMF\) CRU](#)

[Fault-Tolerant Process Communication](#)

[Fault-Tolerant Disk Access](#)

[Maximum Processor Enclosure I/O](#)

[Expansion to Second Processor Enclosure](#)

[Expansion to External I/O Enclosure](#)

[Components of I/O Multifunction \(IOMF\) CRU](#)

[I/O-Oriented System](#)

[Maximum I/O for a Single Processor Enclosure](#)

[Tetrahedral Topology](#)

[First Triangle of Tetrahedron](#)

[Tetrahedral Topology With Four Processor Enclosures](#)

[Maximum I/O for Four Processor Enclosures](#)

[Extending the Tetrahedral Topology](#)

[Maximum System With Single Server](#)

[ServerNet Clusters](#)

[Cluster Topologies for 6770 Switches](#)

[Star Topology](#)

[Split-Star Topology](#)

[Tri-Star Topology](#)

[Layered Topologies for 6780 Switches](#)

[Connections Between Zones in Layered Topology](#)

# Processor Enclosures and I/O Enclosures

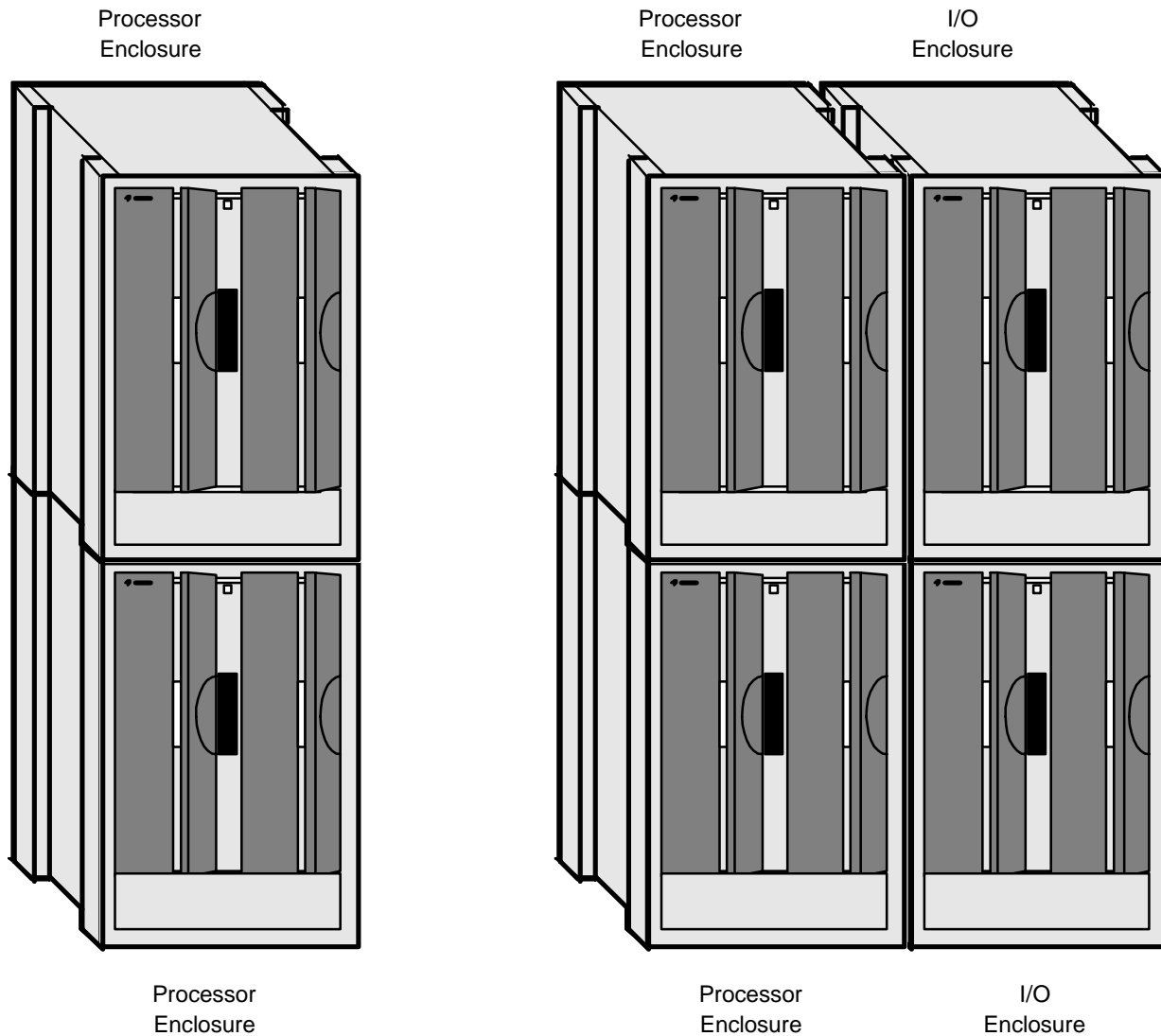
A NonStop S-series server is a computer system that is structured around many parallel components, both physical and logical. For example, a single server can include up to 16 **processors** (CPUs).

Processors are housed two in a **processor enclosure**. Thus a four-processor **server** would be contained in two processor enclosures, as illustrated in the left part of [Figure 1-1](#). As will be shown later, processor enclosures include a considerable I/O capability, so that **I/O enclosures**, illustrated as part of the server in the right part of [Figure 1-1](#) would only be needed where I/O requirements exceed the I/O capability built into the processor enclosures.

Processor enclosures and I/O enclosures are structurally the same. A processor enclosure can be converted to an I/O enclosure (or vice versa). Primarily, the conversion requires swapping the internal **processor multifunction (PMF) CRUs** for **I/O multifunction (IOMF) CRUs**. (CRU is customer-replaceable unit.)

The processor enclosures for NonStop S7000 servers differ from other processor enclosures in one major way. At the bottom of the other enclosures, a shelf houses the power supplies for the processor multifunction (PMF) CRUs. This space is empty for the NonStop S7000 server. The reason for this difference is that the power supplies for the NonStop S7000 processors are part of the processor CRU but are separate for the other processors.

---

**Figure 1-1. A Four-Processor Server Shown Without and With I/O Enclosures**

VST201.vsd

# Maximum Server Configuration

Up to 16 processors can be supported in a single server. With two processors contained in a processor enclosure, a 16-processor system would have eight processor enclosures. Processor enclosures are shown unshaded in [Figure 1-2](#). The shaded boxes represent I/O enclosures.

One processor enclosure can support a theoretical maximum of nine I/O enclosures. However, this theoretical maximum is possible only for a single, stand-alone processor enclosure.

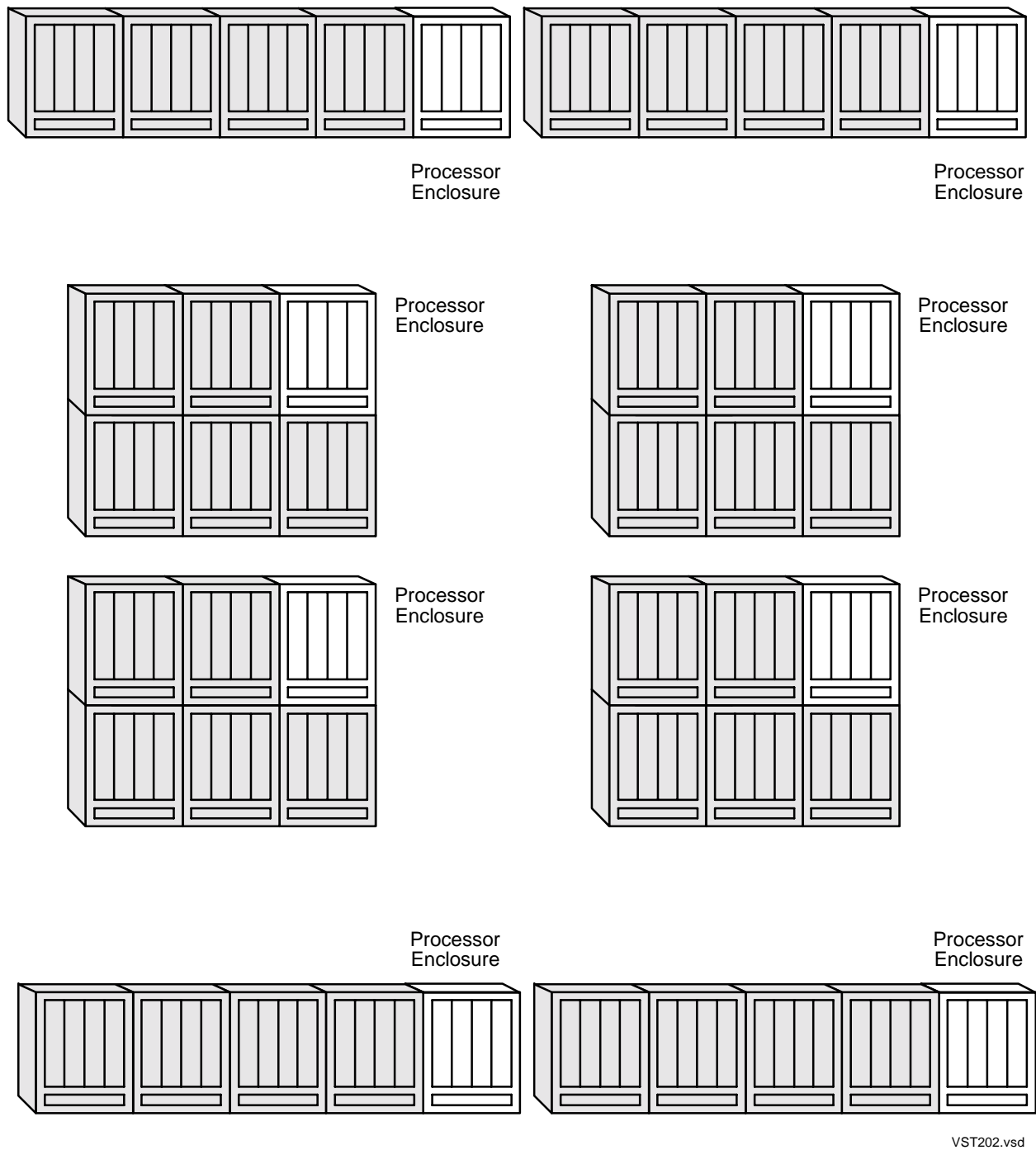
When multiple processor enclosures are interconnected, the number of I/O enclosures that can be supported is reduced. In a maximum configuration, as illustrated in [Figure 1-2](#), each of the four of the processor enclosures forming the **core tetrahedron** is limited to a maximum of five directly connected I/O enclosures. (**Tetrahedron** is a method of processor interconnection that is discussed under [Tetrahedral Topology](#) on page 1-26.) In [Figure 1-2](#), the four processor enclosures (unshaded) in the middle of the diagram are the core tetrahedron.

The outer groupings of enclosures (top and bottom of the figure) also contain processor enclosures. Each of those processor enclosures is connected to the nearest processor enclosure in the core tetrahedron, thus doubling the number of processors that are supported in this maximum configuration. Each of those outer processor enclosures, in turn, can have four I/O enclosures associated with it.

With all possible processor and I/O enclosures present, as in [Figure 1-2](#), the maximum possible server arrangement is 8 processor enclosures and 36 I/O enclosures.



**Figure 1-2. A Maximum Server Would Have 44 System Enclosures**



# Internal Arrangement of System Enclosures

Processor enclosures and I/O enclosures are very similar in their internal arrangement, and the generic term that describes both is **system enclosure**. The primary difference is that, for I/O enclosures, processors do not exist on the CRUs installed in locations 50 and 55. IOMF CRUs are installed in slots 50 and 55 instead. Another difference is that **ServerNet expansion boards** (SEBs) or **modular ServerNet expansion boards** (MSEBs) cannot be installed in I/O enclosures.

---

**Note.** SEBs provide connections that link one system enclosure to another. A modular ServerNet expansion board (MSEB) is a SEB that uses plug-in cards (PICs) to provide a choice of cable media (ECL, fiber-optic, or serial copper) for routing ServerNet packets. In addition, only MSEBs can be used for clustering interconnections. In cases where only MSEBs are permissible, this manual specifically uses the MSEB acronym; otherwise, the SEB acronym is used as a generic term for both kinds of boards.

---

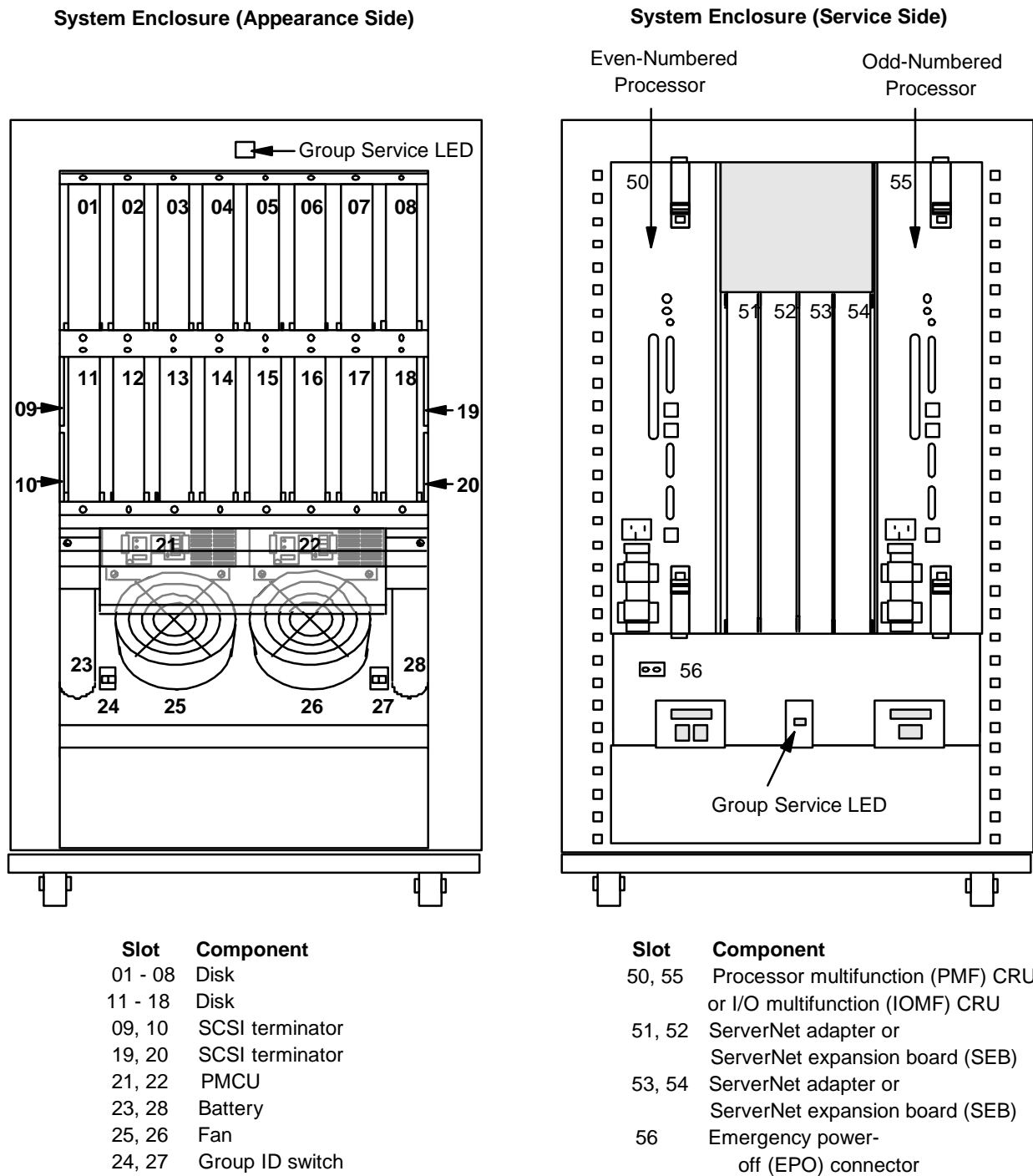
[Figure 1-3](#) illustrates a typical internal arrangement for the system enclosures from both the **appearance side** and the **service side**. (The illustration shows a NonStop S7000 processor enclosure.)

Slots 51 through 54 can contain four **ServerNet adapters**. In the case of a processor enclosure, these slots could instead contain four ServerNet expansion boards, or two of each. (The functions of ServerNet expansion boards are discussed under [Expansion to Second Processor Enclosure](#) on page 1-16.)

Sixteen 3.5-inch disk drives can be accommodated in slots 01 through 08 and 11 through 18, using two internal SCSI buses.

Whether slots 50 and 55 contain processor multifunction (PMF) CRUs (see page [1-8](#)) or I/O multifunction (IOMF) CRUs (see page [1-20](#)), these slots each provide one differential SCSI port for connection to external SCSI devices and one Ethernet port.

Figure 1-3. Processor Enclosure Shown From Both Sides



# Components of Processor Multifunction (PMF) CRU

The **processor multifunction unit (PMF) CRU** consists of two boards and (for NonStop S7000 servers) a power supply. The two boards are the **processor and memory board (PMB)** and the **multifunction I/O board (MFIOB)**. PMF CRUs in NonStop S70000 servers and higher also have a power board.

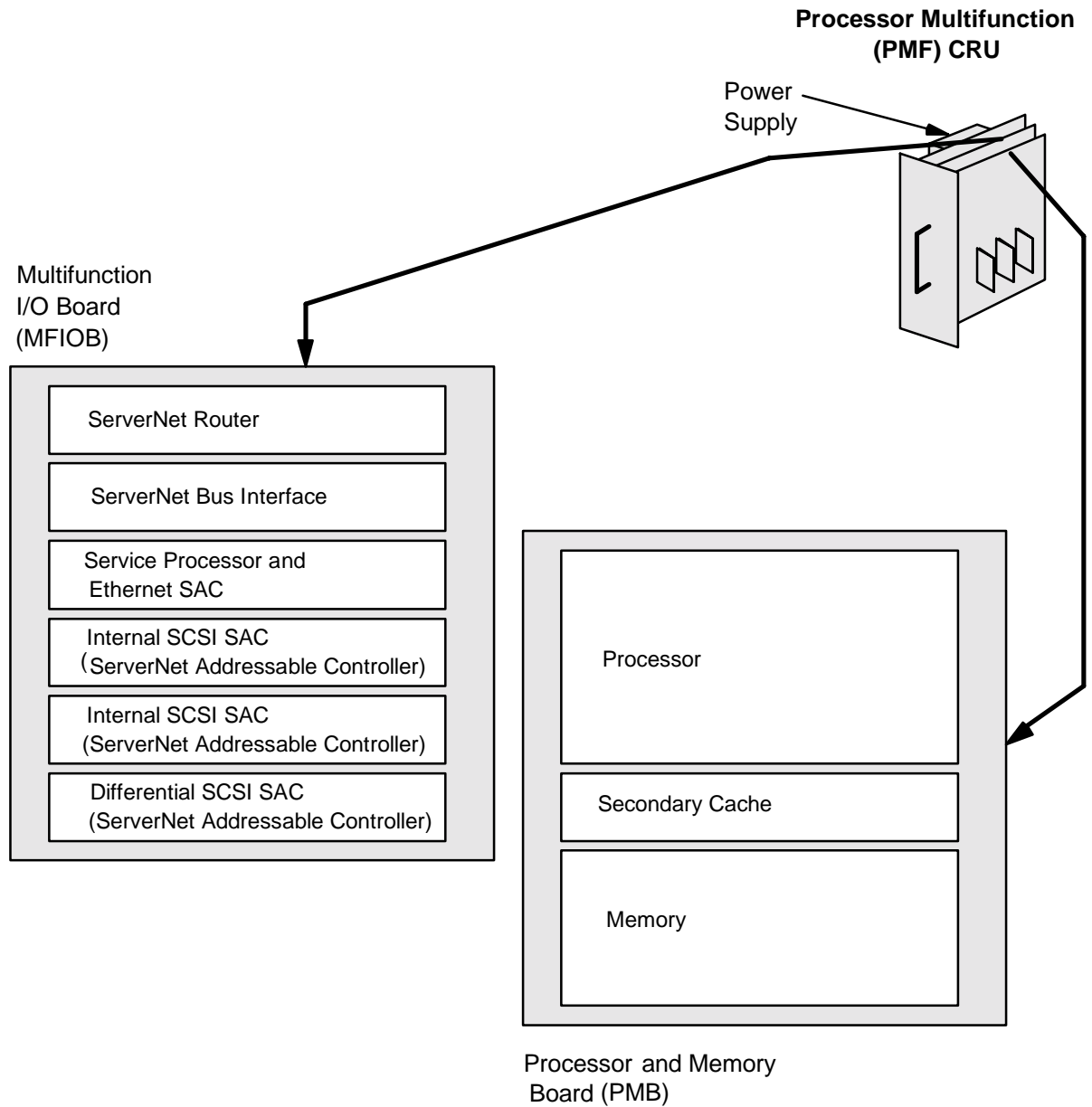
An approximate representation of the PMF CRU is shown in [Figure 1-4](#). The PMF CRU is the type of CRU that is installed in slot 50 and 55 when the enclosure is configured as a processor enclosure, as mentioned in the preceding topic.

The multifunction I/O board has six main logic elements, as indicated in the figure. The **ServerNet router** and the **ServerNet bus interface (SBI)** provide the communication paths into and out of the PMF CRU, both for input/output and for interprocessor communication.

On the MFIOB are four **ServerNet addressable controllers (SACs)**. The first of these provides ServerNet addressability for the Ethernet port and for the **service processor (SP)**. The service processor has interfaces to both the ServerNet hardware and a separate **serial maintenance bus (SMB)**.

The other three SACs on the MFIOB are SCSI interfaces. Two of these connect to the two separate internal SCSI buses, one for each set of eight disk slots. The remaining SAC provides differential connection for an external SCSI device.

The processor and memory board contains the dual, lockstepped MIPS microprocessors, the main memory system, and the secondary cache.

**Figure 1-4. Processor Multifunction (PMF) CRU Has Four ServerNet Addressable Controllers**

VST204.vsd

# Fault-Tolerant Process Communication

**Fault tolerance** for processes is accomplished by providing a **backup process** in some other processor and providing two or more paths of communication between them, so that if one path or processor should fail, the other processor and paths will remain operable. This basic principle of fault-tolerant process communication is illustrated in [Figure 1-5](#).

In the example shown, process A can be assumed to be the **primary process** of a **process pair**. The backup process, process B, is programmed to accept **checkpoint messages** that convey significant changes in the state of the primary process. Upon any failure of the processor that is executing the primary process, the backup process can assume execution of the work from the point of the last valid checkpoint.

Dual data paths between the processes assure that checkpoint messages can still be delivered in the event that one of the paths should fail. Note that process A can send checkpoint messages to process B either through router X or through router Y. Thus if a problem should occur on the MFIOB associated with processor 0, process A can still send its checkpoint messages through the MFIOB associated with processor 1.

Although, for this example, the two processors are represented as being in the same processor enclosure, in fact the fault-tolerant principle applies even if the primary and backup processes are running in processors in different enclosures. The ServerNet architecture is so designed that all X routers are connected together and all Y routers are connected together. Thus process A can send its checkpoint messages to the Y router in its own enclosure and be assured that, through a succession of other Y routers, those messages will reach the backup process no matter where it is.

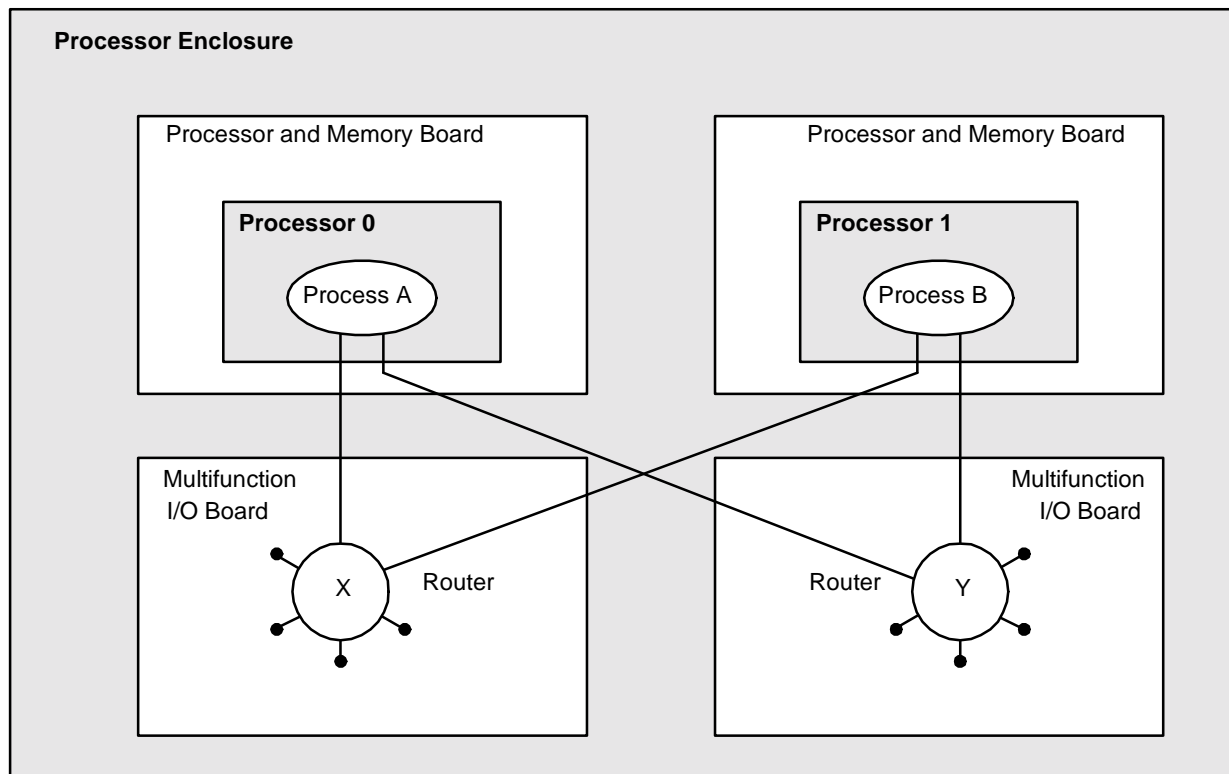
The complex interconnection of all routers into two parallel structures creates two independent data-routing entities, each called a ServerNet **fabric**. The interconnection of the X routers makes up the X fabric, and the interconnection of the Y routers makes up the Y fabric.

---

**Note.** ServerNet routers differ in the number of ports available for routing. For example, SEBs use 6-port routers, whereas MSEBs use 12-port router 2s. PMF CRUs in S7000, S7400, S70000, and S72000 servers use the 6-ported routers, whereas PMF 2 CRUs in S7600, S74000, S76000 and S86000 servers use the 12-ported router 2s. IOMF CRUs use the 6-port routers, whereas IOMF 2 CRUs use the 12-port router 2s. Generally, this manual shows 6 ports for a router unless the illustration pertains specifically to hardware that uses the 12-port router 2s.

---

**Figure 1-5. Processes in Different Processors Have Alternative Communication Paths**



VST205.vsd

## Fault-Tolerant Disk Access

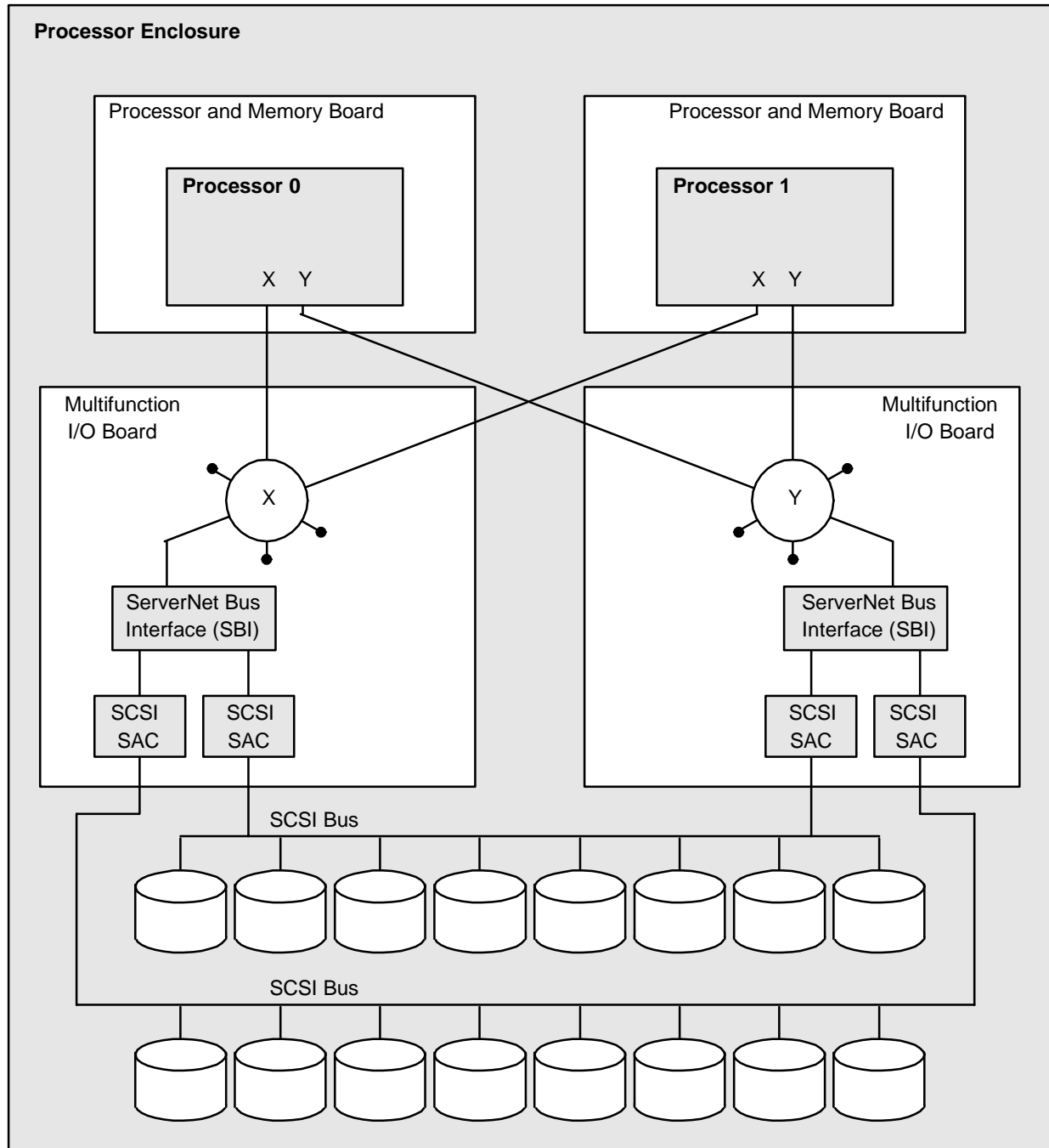
Multiple disk drives are provided for within the processor enclosure. A single SCSI bus can support up to eight of these drives, and there are two such buses.

Both processors in the enclosure have fault-tolerant access to all installed drives. That is, either processor can access the drives through either of the two multifunction I/O boards (MFIOBs).

[Figure 1-6](#) illustrates the basic I/O configuration for the two SCSI buses. Note that one MFIOB provides access through the X router, and the other MFIOB provides access through the Y router.

In addition to fault-tolerant disk access paths, fault tolerance for databases can be augmented by the use of **mirrored volumes**. This optional access method maintains duplicate data on two separate physical volumes. Thus, whenever data is read from the files, either volume can be accessed because they contain identical information. All data written out to the files is automatically written on both disk volumes.



**Figure 1-6. In Any Processor Enclosure, Both Processors Can Access All Disks**

VST206.vsd

# Maximum Processor Enclosure I/O

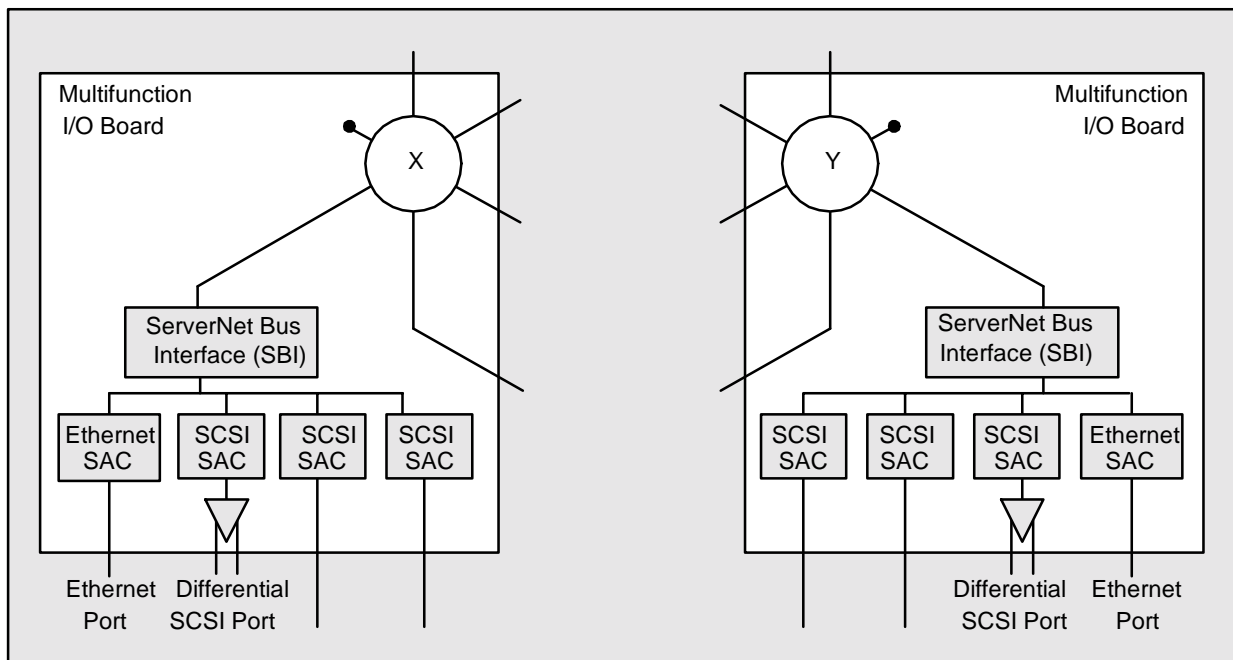
A single processor enclosure can support 16 disk drives using the two SCSI buses, and in addition provides two differential SCSI ports, two Ethernet ports, and any I/O devices connected to two ServerNet adapters. [Figure 1-8](#) shows this maximum configuration for one processor enclosure.

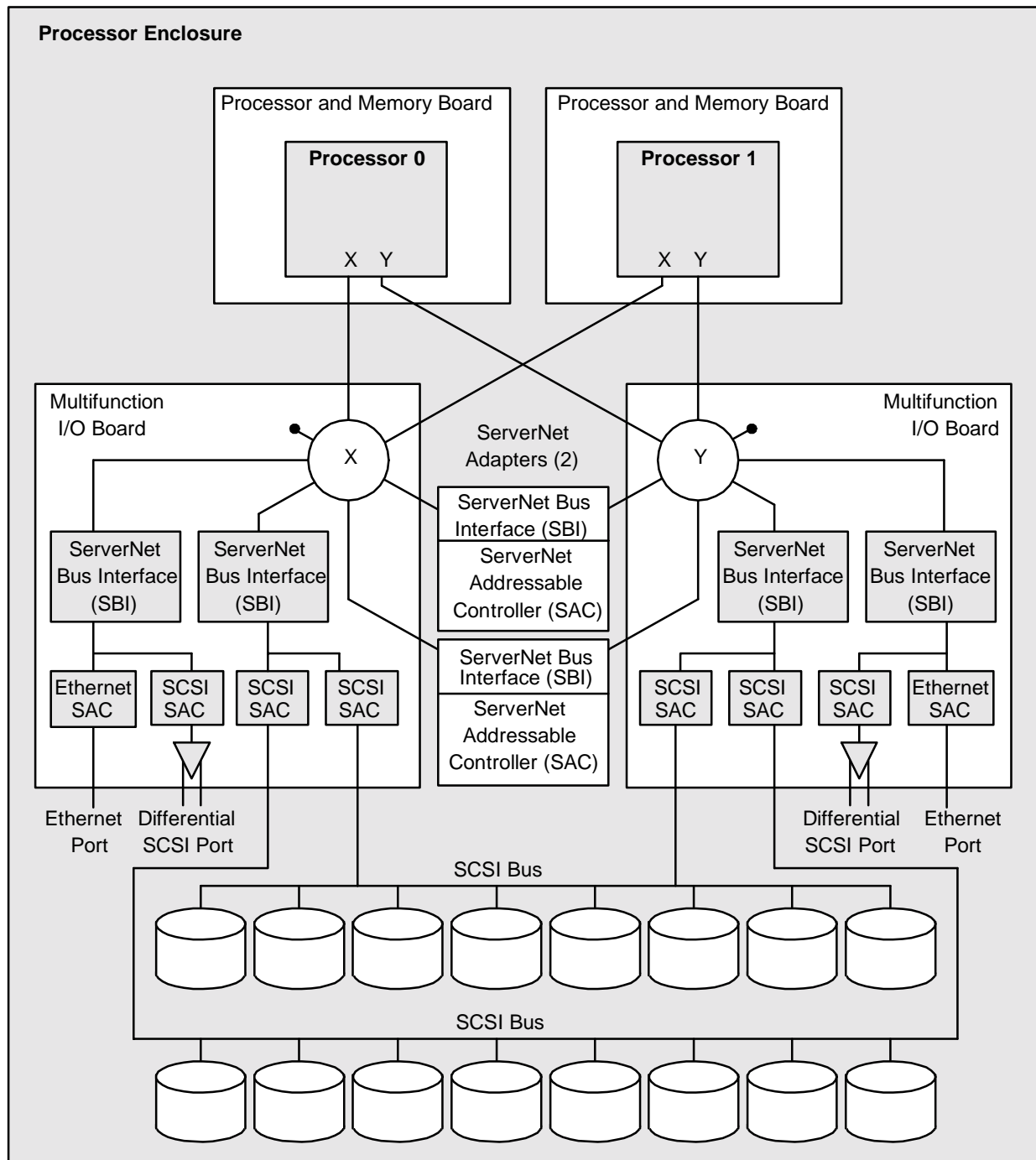
There is a slight difference in the way this I/O capability is managed in different server models. In the case of the NonStop S7600, S74000, S76000, and S86000 servers, an additional router port is available and is used to provide a separate ServerNet bus interface (SBI) for the disk drives on the SCSI buses. In the case of NonStop S7000, S7400, S70000, and S72000 servers, the disk drives are managed through the same SBI as the Ethernet ports and differential SCSI ports, as shown in [Figure 1-7](#).

ServerNet adapters always have a ServerNet bus interface (SBI) for connecting to the ServerNet routers and one or two ServerNet addressable controllers (SACs). The SAC is specialized to the type of input/output service provided, such as ATM communications. As shown in [Figure 1-3](#) on page 1-7, ServerNet adapters are installed in slots 51 through 54.

Note that one unused router port is available in each of the MFIOBs. This port is reserved for expanding the ServerNet fabric. See [Expansion to Second Processor Enclosure](#) on page 1-16.

**Figure 1-7. Alternate Configuration of MFIOBs for NonStop S7000, S7400, S70000, and S72000 Servers**



**Figure 1-8. Dual MFIOBs Provide Large I/O Capability for a Processor Enclosure**

VST207.vsd

# Expansion to Second Processor Enclosure

A pair of **ServerNet expansion boards (SEBs)** can be used to connect one processor enclosure to another. [Figure 1-9](#) shows the arrangement of connections.

The main logic component of an SEB is a router that is additional to the one already present on the MFIOB. SEBs are always added in pairs because one is needed for the X fabric and one is needed for the Y fabric.

In each case, the router connection from the MFIOB that connects to an SEB is the router port that was shown unused in [Figure 1-7](#) and [Figure 1-8](#). The four downward-pointing arrows from the MFIOB routers connect to the SCSI bus and I/O slots like the example shown in [Figure 1-8](#) on page 1-15.

---

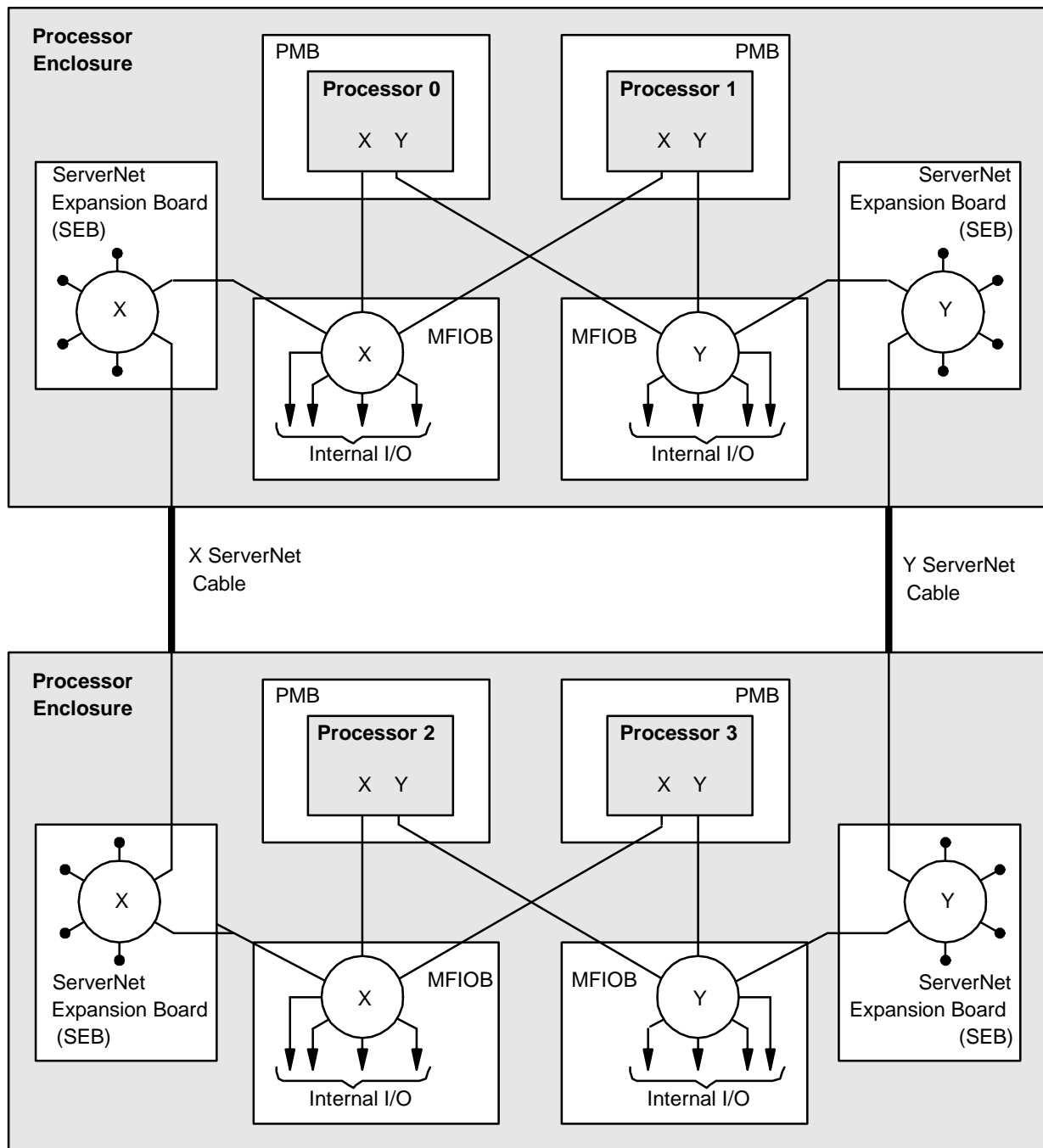
**Note.** For clarity, the I/O details are omitted in [Figure 1-9](#). As explained in the preceding topic ([Maximum Processor Enclosure I/O](#)), I/O configuration details vary. For example, because router 1s have only six ports, slots 51 and 52 are single-ported while slots 53 and 54 are dual-ported. Router 2s, on the other hand, have 12 ports, and so slots 51 and 52 are also dual-ported. NonStop S7000, S7400, S70000, and S72000 servers use three, rather than four, router ports for internal I/O logic.) Two of these ports are for the two ServerNet adapters, and the remaining two (or one) are for the SCSI buses, Ethernet port, service processor, and differential SCSI port.

---

The required cables for interconnecting the two processor enclosures are shown with bold lines in the example in [Figure 1-9](#). One cable interconnects the X routers on one SEB in each of the processor enclosures, and the other cable interconnects the Y routers.

Note that four unused router ports are available on each of the SEBs. These ports can be used for expanding either, or both, the number of processor enclosures and the number of I/O enclosures. Expansions of both kinds are the subject of the remaining topics in this section.

**Figure 1-9. Processor Enclosures Are Interconnected Through ServerNet Expansion Boards**



VST208.vsd

## Expansion to External I/O Enclosure

ServerNet expansion boards (SEBs) provide router ports that can expand to I/O enclosures as well as to other processor enclosures (as in the preceding topic).

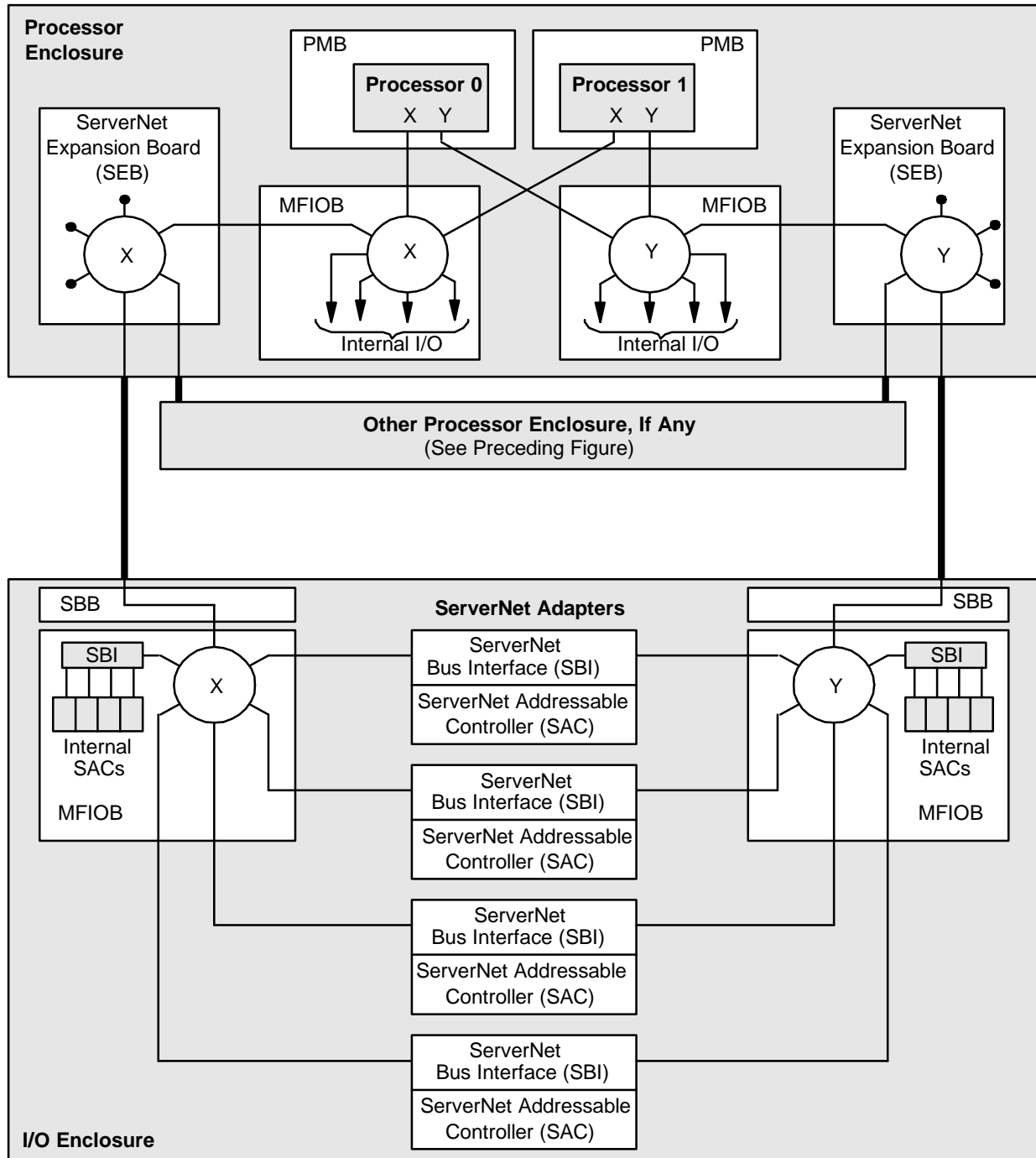
[Figure 1-10](#) shows expansion from one processor enclosure to one I/O enclosure. In this case, three of the SEB router ports are used, and three are unused.

[Figure 1-10](#) assumes that the second processor enclosure is still connected to the same router ports (X and Y) of the first processor enclosure. However, the internal details of that second enclosure are not shown here.

As before, the ServerNet cables between enclosures are represented by bold lines. Note that, in the I/O enclosure, the cable does not connect to another SEB but instead the ServerNet lines are routed through a **ServerNet buffer board** (SBB) to a multifunction I/O board (MFIOB). (SEBs are not supported in I/O enclosures.) The SBB physically does not occupy the position of an SEB but rather is part of the I/O multifunction (IOMF) CRU, as shown in the next topic.

The internal arrangement of the X and Y routers in the I/O enclosure is similar to that in a processor enclosure, but only six-port routers are currently supported.

However, note that, because there are no router connections to processors, two additional router ports are available for I/O connections. (Compare MFIOBs in the two enclosures.) Five of the MFIOB router ports are available for internal I/O. The two additional ports are used for two additional ServerNet adapters, so that each I/O enclosure can support four ServerNet adapters, rather than just two in the case of processor enclosures.

**Figure 1-10. ServerNet Expansion Boards Also Can Connect to I/O Enclosures**

VST209.vsd

## Components of I/O Multifunction (IOMF) CRU

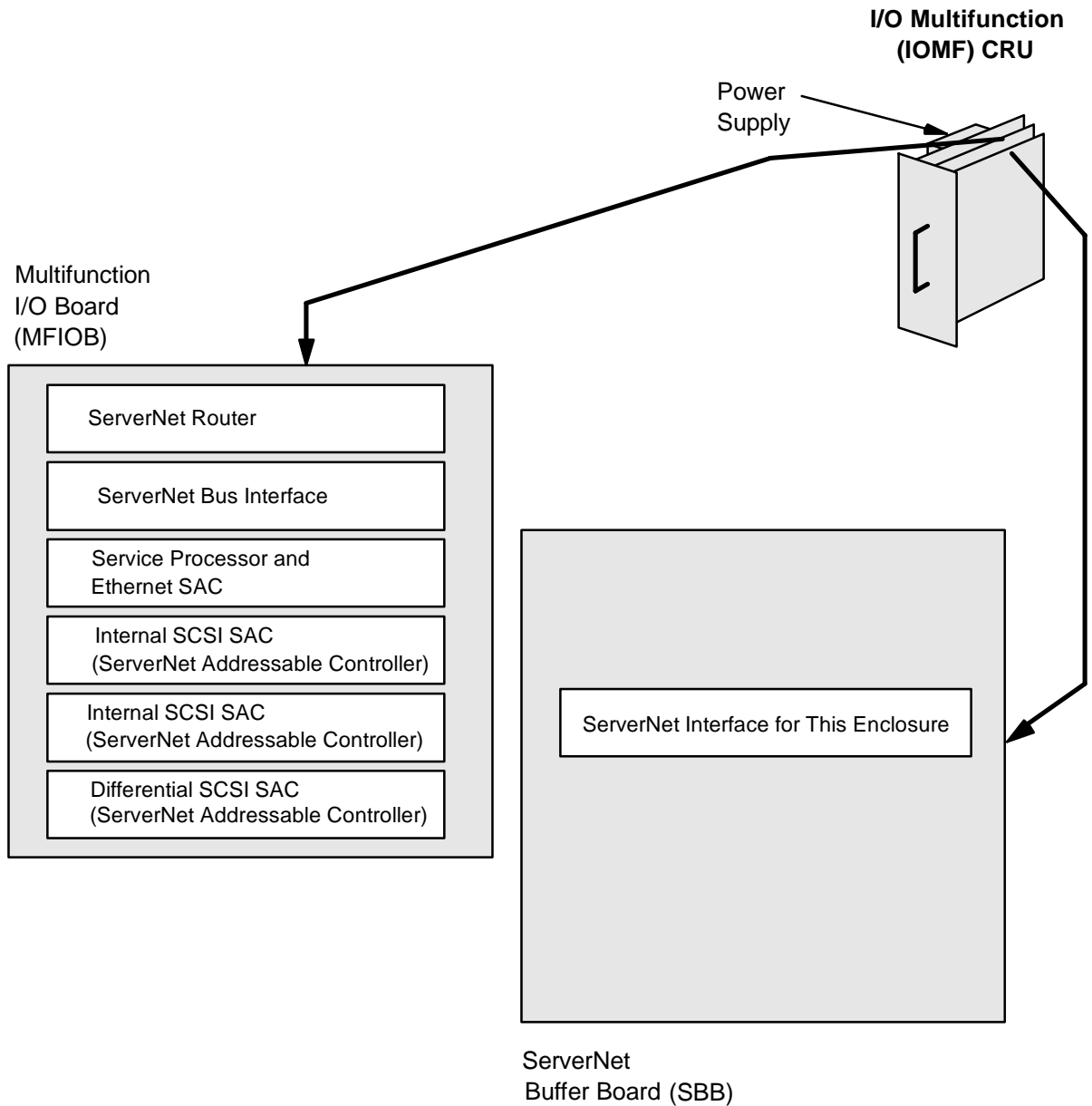
The I/O multifunction (IOMF) CRU consists of two boards and a power supply. The two boards are the multifunction I/O board (which is identical to that shown earlier for the PMF CRU) and the ServerNet buffer board (SBB).

An approximate representation of the IOMF CRU is shown in [Figure 1-11](#). The IOMF CRU is the type of CRU that is installed in slots 50 and 55 when the enclosure is configured as an I/O enclosure, as shown in [Figure 1-3](#).

As mentioned in [Components of Processor Multifunction \(PMF\) CRU](#) on page 1-8, the MFIOB has six main logic elements: the ServerNet router, the ServerNet bus interface, and four ServerNet addressable controllers.

The ServerNet buffer board contains logic for sending and receiving ServerNet packets to and from other enclosures. It provides the necessary ServerNet cable connectors.



**Figure 1-11. I/O Multifunction (IOMF) CRU Includes External Enclosure Interface**

VST210.vsd

## Multiple I/O Enclosures

When two processor enclosures are interconnected in a basic configuration (using a single pair of SEBs), they can each support two I/O enclosures. The resulting six system enclosures are connected as shown in [Figure 1-12](#). In this case, the router ports from the SEBs serve two functions: they interconnect the ServerNet fabrics in the two processor enclosures, and they provide external I/O access to two I/O enclosures.

In [Figure 1-12](#), smaller blocks represent logic that was shown in detail in previous topics. For example, the entire internal I/O structure for processor cabinets, shown in detail in [Figure 1-8](#) on page 1-15, is represented by a pair of ServerNet adapter blocks and a pair of blocks divided into four parts that represent the four internal ServerNet addressable controllers (SACs).

---

**Note.** For simplicity, [Figure 1-12](#) and [Figure 1-13](#) show six-port routers in association with the processors, thus resembling the configuration shown in [Figure 1-7](#) on page 1-14. As shown in [Figure 1-8](#) on page 1-15, other NonStop servers have seven implemented router ports (instead of six) and use two of them (instead of just one) to service the four internal SACs.

---

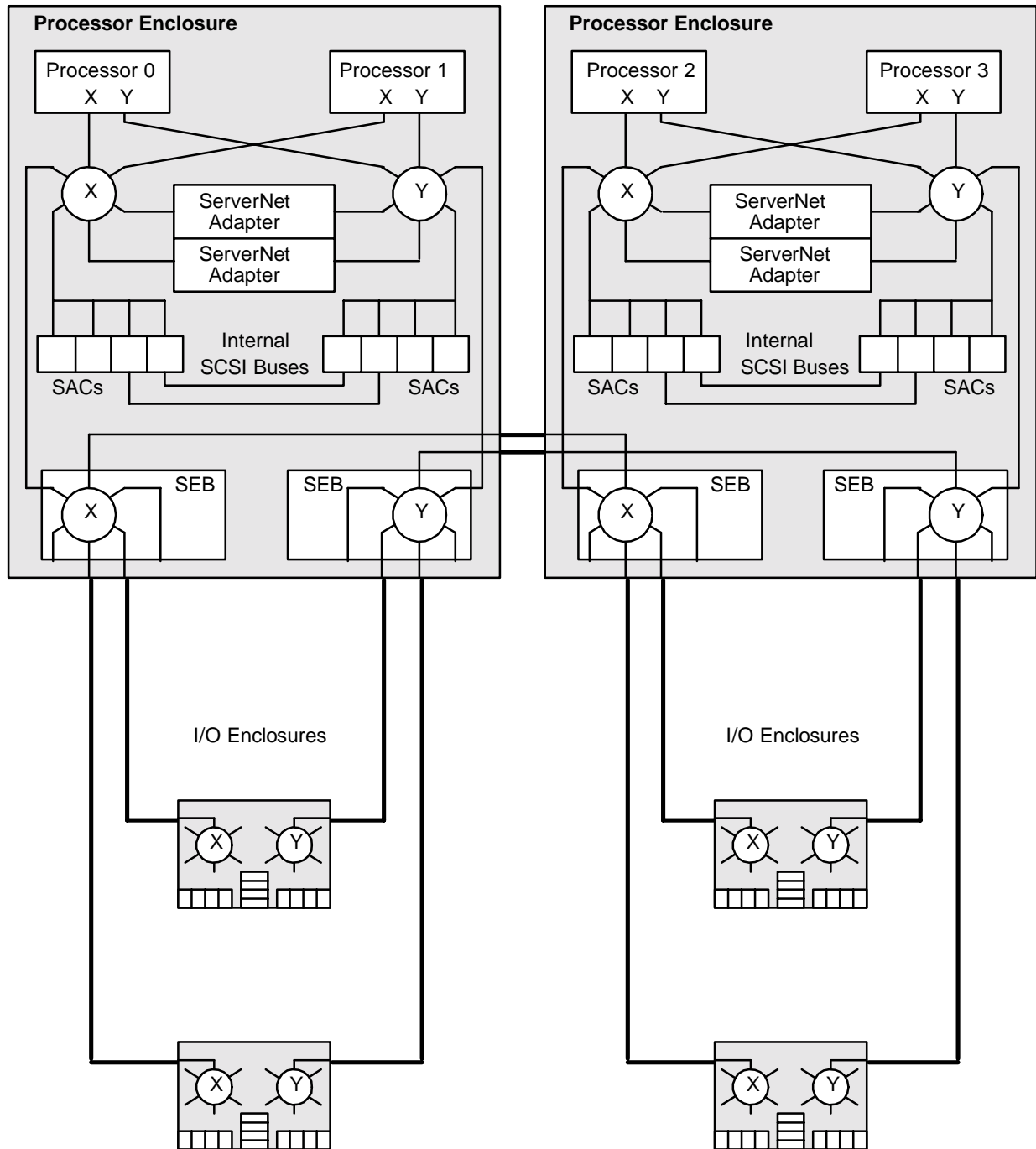
The I/O enclosure logic, shown earlier in [Figure 1-10](#) on page 1-19, is even further miniaturized to show eight enclosures in [Figure 1-12](#). The only router connections actually shown are those for the external connections to the routers in the SEBs in the processor enclosures. The remaining five connections of each router are as previously shown: one for each side of the four available ServerNet adapters (middle stack) and the fifth one for the ServerNet bus interface (SBI, not shown) that interfaces to the four SACs that are on each multifunction I/O board (MFIOB). Those four SACs, for the SCSI bus, Ethernet, and so on, are represented by the blocks in the lower left and right corners of the I/O enclosures.

In operation, note that each processor has two paths (either X or Y) to access any SAC (which includes the ServerNet adapters) anywhere in [Figure 1-12](#). That is possible because of the SEB-to-SEB connection between the two processor enclosures. Each passage through a router is a “router hop.” The **hop count** can vary from one to four in this configuration.

Also note that it is not possible to cross over between X paths and Y paths. No matter how extensive the ServerNet structure grows, the ServerNet fabrics for X and Y remain separate and independent. Both are in simultaneous use at all times. A failure of a **ServerNet link** between routers (or between a router and a SAC) affects only that portion of the fabric. The remainder of the fabric remains fully functional. Operating system software detects such link failures and provides automatic rerouting through the opposite fabric whenever necessary.

In [Figure 1-12](#), the configuration shown has ServerNet adapters in the processor enclosures. However, in your own computing environment, those ServerNet adapters can be replaced with pairs of additional SEBs instead. Such a configuration provides much more I/O capability, as illustrated next.

**Figure 1-12. Any Processor Has Dual Access Paths to All I/O, External or Internal**



VST211.vsd

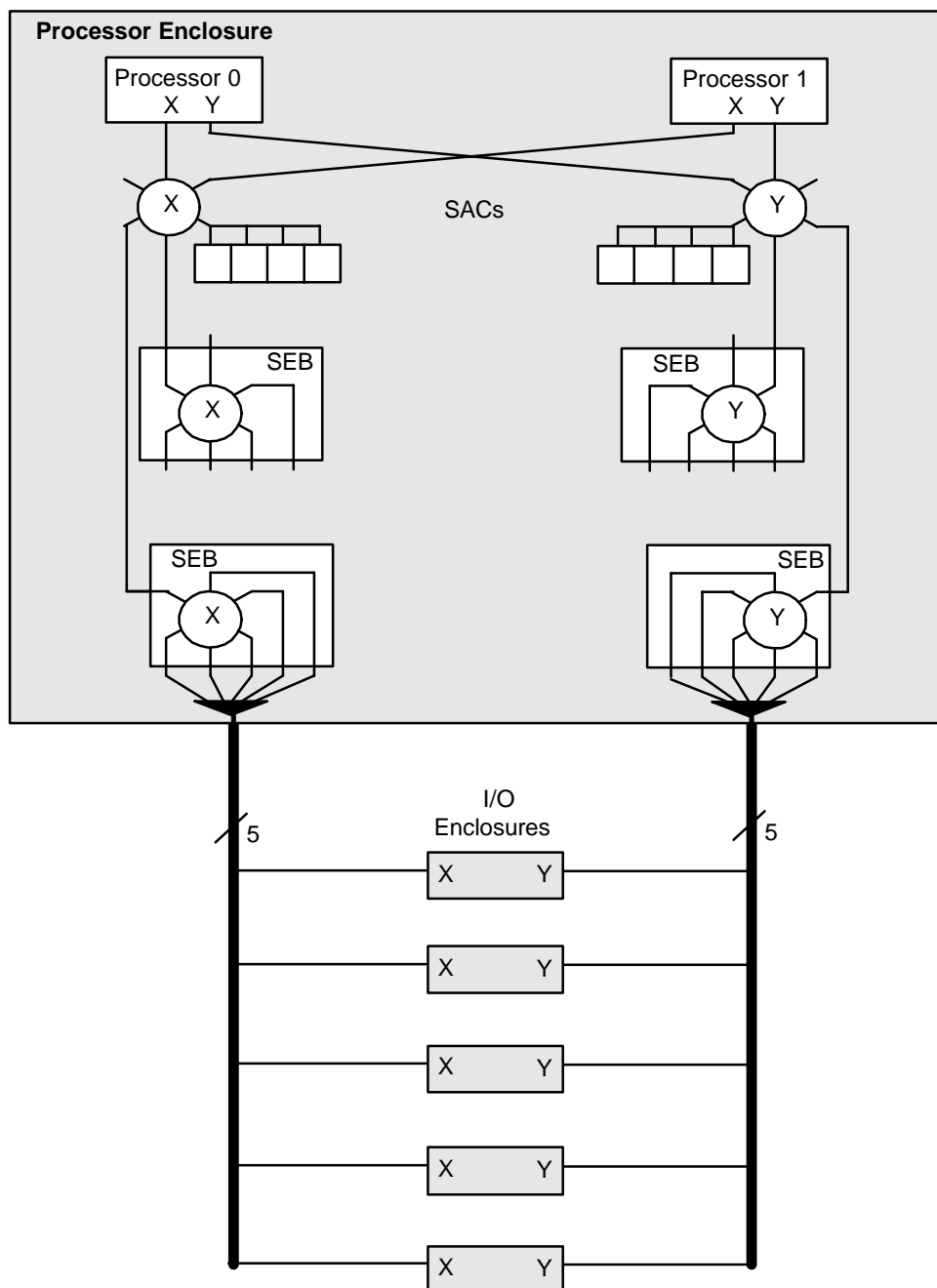
## Maximum I/O for a Single Processor Enclosure

If ServerNet adapters are excluded from the processor enclosures, two additional ServerNet expansion boards can be installed, one for the X fabric and one for the Y fabric. As shown in [Figure 1-13](#), these additional SEBs increase the number of I/O enclosures that can be supported from two to five.

In comparison with [Figure 1-12](#), note that the processor enclosure still has the four ServerNet addressable controllers (SACs) for internal I/O. However, the processor enclosure now has four SEBs instead of two, and no ServerNet adapters.

The added SEBs (lower row of SEBs across the diagram) have five links available for I/O enclosure connection, whereas the original SEBs (upper row) have no links for I/O enclosures. When dual SEBs are present, their functions are separately defined. One pair of SEBs is used strictly for connection to other processor enclosures, and the other pair is used strictly for connection to I/O enclosures.

In [Figure 1-13](#), some ServerNet links have been grouped together for drawing convenience. Such groups of links should be understood to be electrically and physically separate. Bold lines are used to represent the fact that several links are present. A slash through the bold line and an accompanying numeral designate the number of links. A black triangle shows the starting point of such groupings.

**Figure 1-13. Dual SEBs Permit Connection of More I/O Enclosures**

VST212.vsd

# Tetrahedral Topology

When more processor enclosures are added to a system, the core of the topology is designed as a **tetrahedron**, idealized in the upper part of [Figure 1-14](#). This design, called **tetrahedral topology**, is used to minimize the number of router hops needed to get from the processors in one enclosure to those in another.

The routers illustrated here are only those on the ServerNet expansion boards (SEBs). Remember that there also are routers on the multifunction I/O boards (MFIOBs) associated with all of the processors; see [Figure 1-9](#) through [Figure 1-13](#).

In the ideal case, any processor can communicate with any other by going through zero or two SEB routers (zero if in the same group of four). However, the ideal case is not possible with a six-port router, because seven ports would be needed for each point of the tetrahedron.

The actual arrangement used is shown in the lower part of [Figure 1-14](#). In this case, the first eight processors, numbered 0 through 7, connect directly to those routers at the four points of the tetrahedron—two per router. That accounts for two ports of each router. With three ports needed to connect to the other routers forming the tetrahedron, one port is left to extend the configuration. This port provides a single link to another router, which provides connection for processors numbered 8 through 15.

Using single SEB pairs in the processor enclosures (as illustrated in [Figure 1-12](#) on page 1-23), the maximum configuration possible is limited to the core tetrahedron. That configuration is limited to eight processor enclosures and is therefore termed the **Tetra 8 topology**. Because each processor enclosure in a Tetra 8 configuration can support two I/O enclosures, Tetra 8 servers are limited to a maximum of eight I/O enclosures. Additionally, Tetra 8 servers cannot participate in ServerNet clusters.

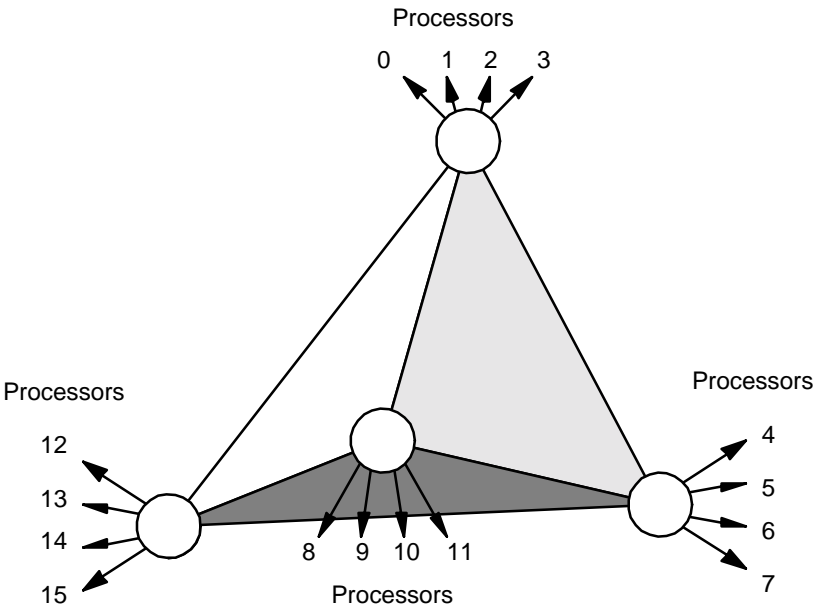
Using dual SEB pairs in the processor enclosures (as illustrated in [Figure 1-13](#) on page 1-25), the maximum configuration permits 16 processor enclosures and is therefore termed the **Tetra 16 topology**. Tetra 16 servers can have up to 36 I/O enclosures and can participate in ServerNet clusters for much larger configurations. See [ServerNet Clusters](#) on page 1-36.

The normal growth pattern for Tetra 16 servers is to complete the core tetrahedron first and then expand on each corner.

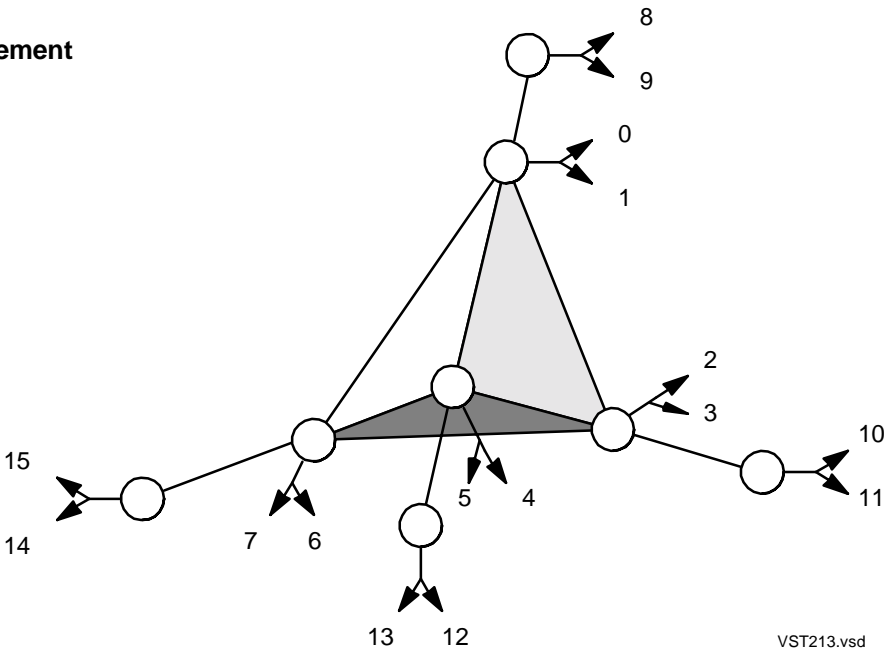
[Figure 1-15](#) to [Figure 1-19](#) show how the conceptual arrangement illustrated in this topic is implemented with physical ServerNet connections.

Figure 1-14. Tetrahedral Topology Efficiently Connects Processor Enclosures

Ideal  
Case



Actual Arrangement



VST213.vsd

# First Triangle of Tetrahedron

[Figure 1-15](#) illustrates the first step in constructing the tetrahedral topology. That first step is to complete a triangle of processor enclosures.

For simplicity in [Figure 1-15](#), only the relevant routers are shown. These routers are the ones on the ServerNet expansion boards (SEBs). The upward-pointing arrows from the routers are connections to the two internal processors in each processor enclosure. However, be aware that this connection is through another router, the router that provides ServerNet links to the internal I/O (refer back to [Figure 1-12](#) and [Figure 1-13](#)).

The top part of [Figure 1-15](#) represents the two-processor enclosure, Tetra 8 system shown in [Figure 1-12](#). Remember that it is possible to connect only two I/O enclosures to each of the processor enclosures.

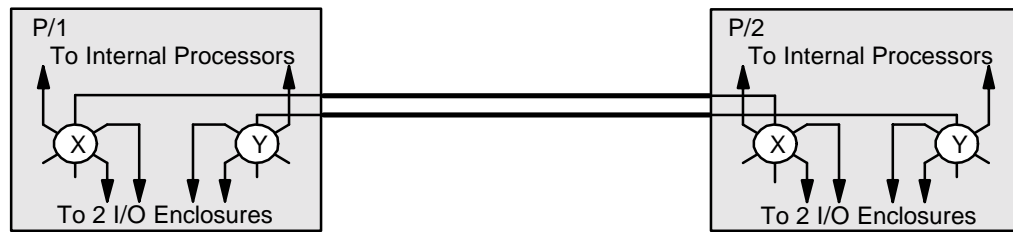
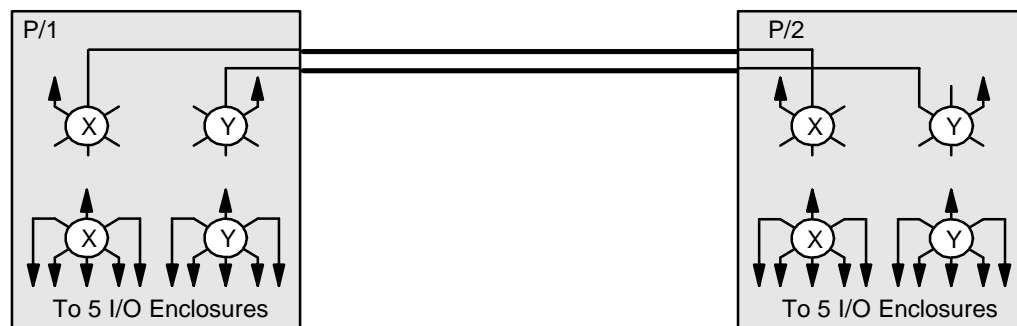
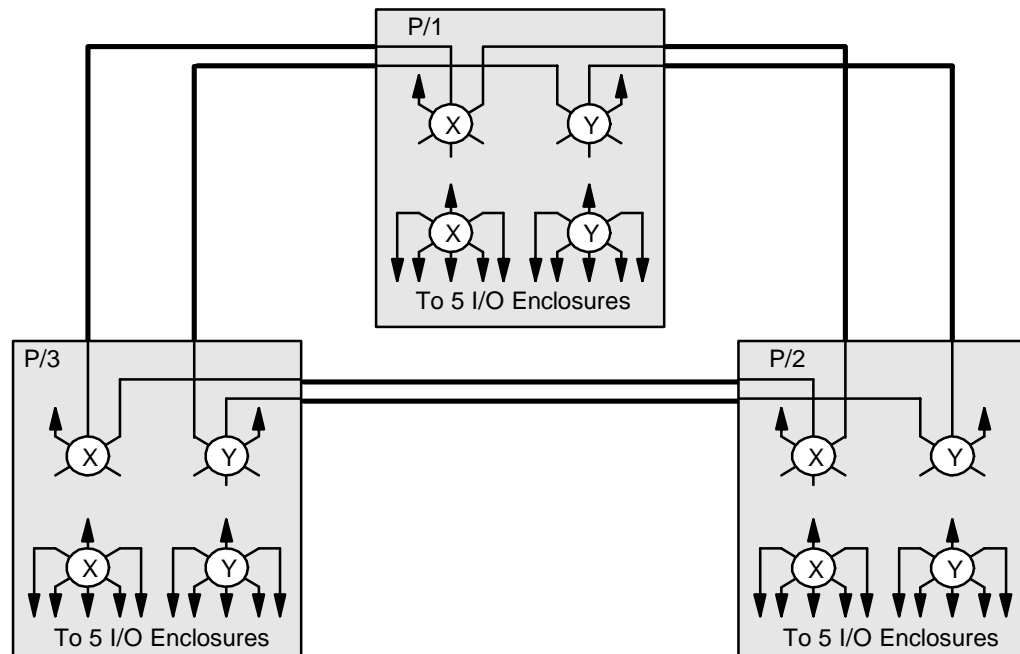
The middle part of [Figure 1-15](#) shows the Tetra 16 version of a two-processor enclosures system. Note that the I/O capability is increased to ten I/O enclosures.

The lower part of [Figure 1-15](#) shows the triangle completed for both X and Y fabrics, using Tetra 16 topology. The added routers for these SEBs provide connections for a total of 15 I/O enclosures. The original pair of SEBs is restricted to implementing the triangle or tetrahedron of processor enclosures (and expansion to [ServerNet Clusters](#)), and is not available for connection to I/O enclosures.

In this and succeeding figures, processor enclosures are numbered with the designations P/1, P/2, and so on. For configuration purposes, these mostly equate to a group number, which generally means all the CRUs in a system enclosure; however, the exact meaning of a **group** is defined by the service processor.

The next step, expanding from three processor enclosures to four, adds the extra dimension that allows completion of the tetrahedron. That step is described in the next topic.



**Figure 1-15. Tetra 16 Configuration Extends Server Expansion****Two Processor Enclosures  
Tetra 8****Two Processor Enclosures  
Tetra 16****Three Processor Enclosures  
Tetra 16**

VST214.vsd

# Tetrahedral Topology With Four Processor Enclosures

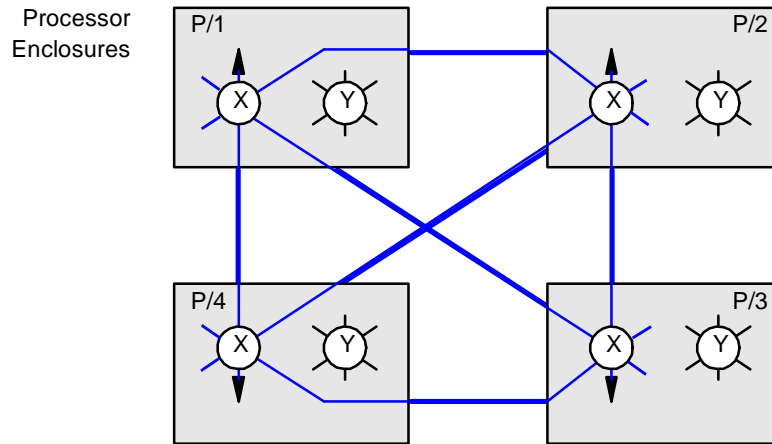
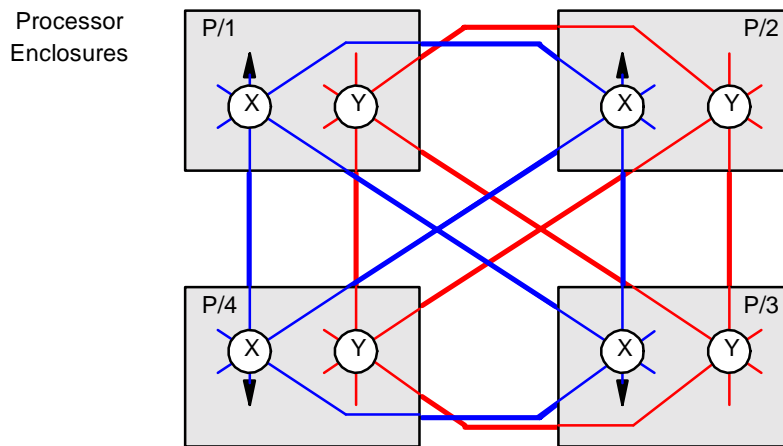
For expansion to a fourth processor enclosure, the enclosures now require full connection as a tetrahedron. The tetrahedral topology maintains the requirement of each processor enclosure having direct access to any other processor enclosure in the configuration.

The cabling depiction for tetrahedral topology is presented in a standard form in [Figure 1-16](#). The “X” pattern of links in the center of this configuration, plus the outer square, is equivalent to the three-dimensional drawing shown in [Figure 1-14](#). In both styles of drawing, each corner router connects to the other three.

Processor enclosures are designated as P/1 through P/4.

The upper part of the figure shows only the X fabric connected, for simplicity. The lower part shows both fabrics connected. In succeeding diagrams, the connections for the Y fabric are omitted to reduce complexity. In all cases, the Y-fabric pattern is identical to what is shown for X-fabric connections.

As previously mentioned, the routers shown are only those on the ServerNet expansion boards (SEBs).

**Figure 1-16. Adding Fourth Processor Enclosure Completes the Core Tetrahedron****X Fabric Alone****Both Fabrics**

VST215.vsd

## Maximum I/O for Four Processor Enclosures

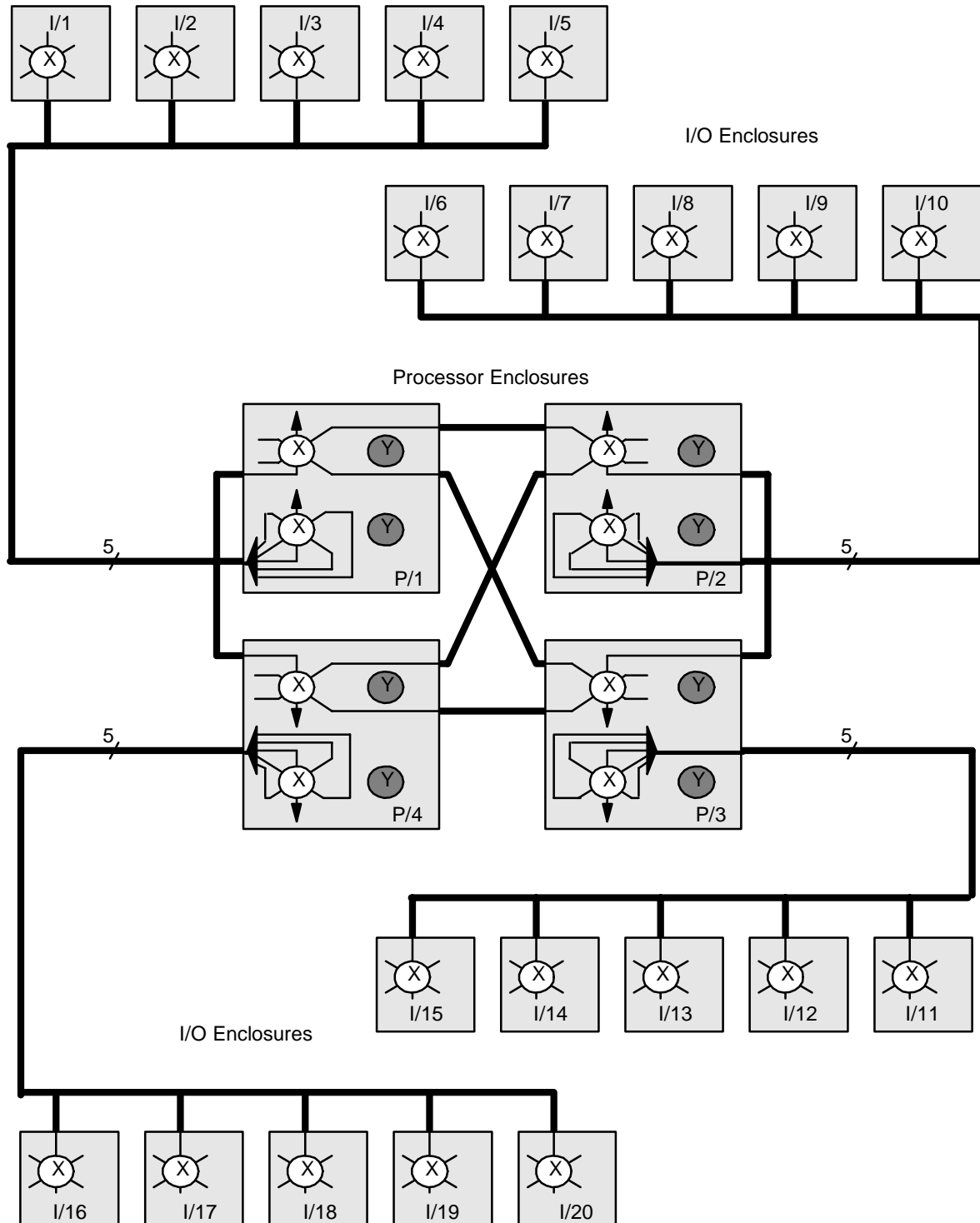
In tetrahedral topologies, two SEBs for each fabric are required in order to include the I/O enclosures. The arrangement is as shown in [Figure 1-17](#). In this case, the four SEB routers that implement the tetrahedron are not available for connection to I/O enclosures. However, the added four SEBs each provide for connection of five I/O enclosures. With all of the available router ports connected, 20 I/O enclosures can therefore be supported.

The I/O enclosures are designated I/1 through I/20, just as the processor enclosures are designated P/1 through P/4.

All of the I/O enclosures can support up to 18 ServerNet addressable controllers (SACs) as shown in [Figure 1-10](#) on page 1-19: five on each multifunction I/O board (MFIOB) and one or two on each ServerNet adapter.

The arrows in the processor enclosures designate router connections to the processors.

For simplicity, the Y fabric connections are not shown.

**Figure 1-17. With Dual SEBs, Core Tetrahedron Supports 20 I/O Enclosures**

VST216.vsd

## Extending the Tetrahedral Topology

As processor enclosures are added, the ultimate plan for the server as a whole is to grow toward the configuration shown in [Figure 1-18](#).

The first four processor enclosures would be configured as a tetrahedron, like the middle four enclosures shown in [Figure 1-18](#). Expansion would then occur from any of the four corners.

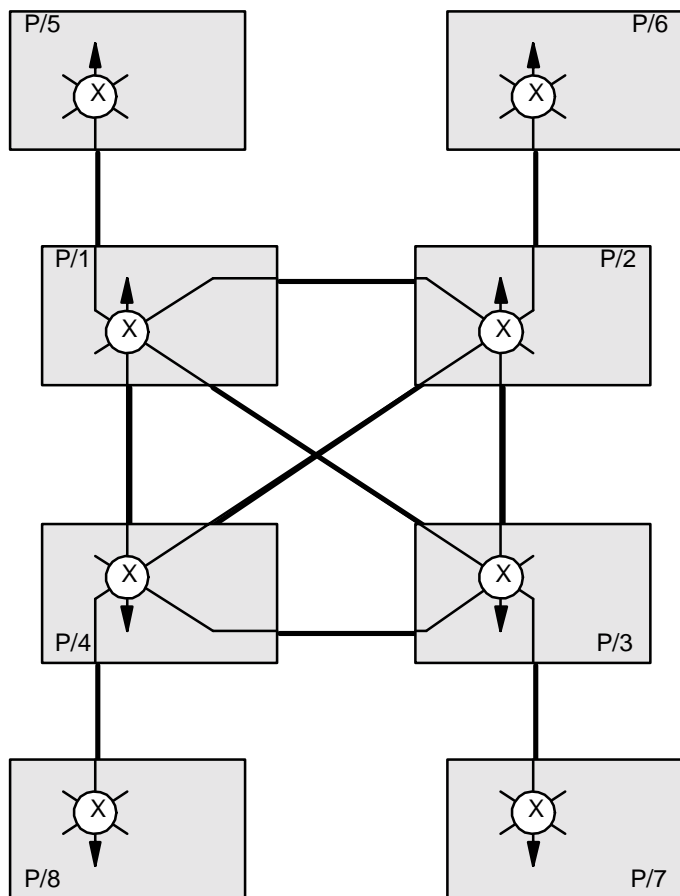
The actual sequence of adding processor enclosures can vary.

Each connection from the core tetrahedron to a processor enclosure requires one router port on a SEB. The addition of I/O enclosures requires additional SEBs, as described in the next topic, [Maximum System With Single Server](#).)

Because this extension of the configuration makes it possible to grow to 16 processors, this configuration is called the Tetra 16 topology.

In comparison, the arrangement of including only the core tetrahedron limits the extension of the configuration to 8 processors, and is therefore called the Tetra 8 topology; see [Figure 1-16](#) on page 1-31.

---

**Figure 1-18. Processor Enclosures Exceeding Four Are Added to Corners**

VST217.vsd

# Maximum System With Single Server

As shown previously in [Figure 1-17](#) on page 1-33, five I/O enclosures can be connected to the routers in the core tetrahedron. When more processor enclosures are added to those of the core tetrahedron, those additional, outer processor enclosures can each have four I/O enclosures.

With all processor enclosures added as shown in [Figure 1-18](#), and with all I/O enclosures connected, the configuration would be as shown in [Figure 1-19](#). In this case, 36 I/O enclosures can be supported.

## ServerNet Clusters

Still referring to [Figure 1-19](#), because no router ports in the core tetrahedron routers are used for connection to I/O enclosures, the availability of these ports makes it possible to interconnect multiple servers in a **cluster**. The clustering of servers permits expansion of the overall system far beyond the limits of a single server. The remaining topics in this section describe the various methods of clustering servers.

Just as the processors and I/O enclosures within a single server can have different interconnection topologies, so also can the interconnection of *clusters* have different topologies. Four such topologies are currently supported for ServerNet clusters:

- Star topology
- Split-star topology
- Tri-star topology
- Layered topology

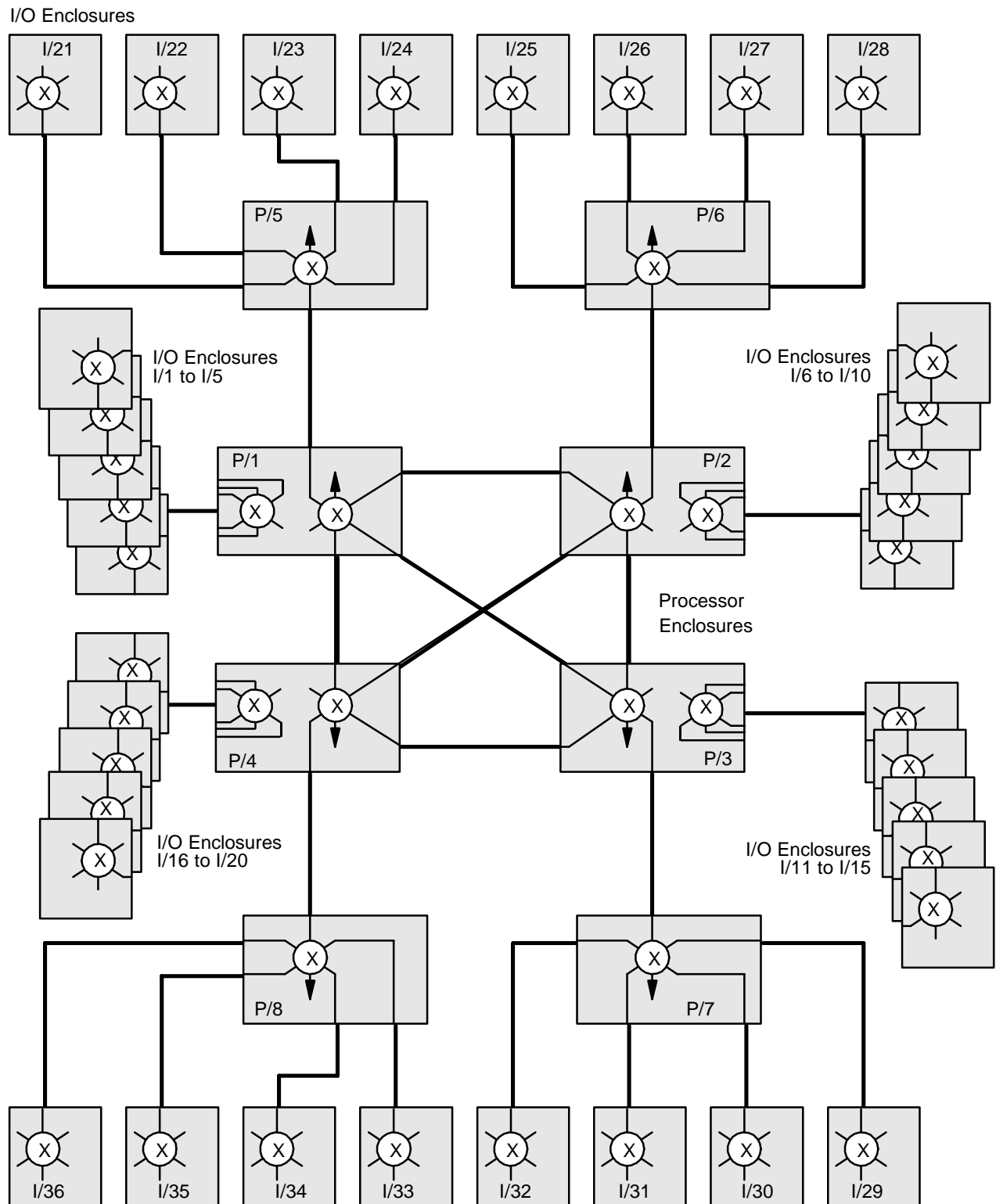
The first three of these topologies are connective variations that are made possible by the use of HP NonStop Cluster Switches (model 6770). The layered topology, which provides the greatest extent of clustering capability, requires the use of HP NonStop ServerNet Switches (model 6780). The two models of cluster switches cannot coexist on the same external ServerNet fabric.

---

**Note.** A cluster is a collection of servers, or nodes, that can function either independently or collectively as a processing unit. This use of “cluster” differs from the definition of a cluster in a FOX ring. In a FOX ring, a cluster is synonymous with “server” and refers to a collection of processors and I/O devices rather than a collection of servers.

---



**Figure 1-19. Eight Processor Enclosures Can Support 36 I/O Enclosures**

VST218.vsd

# Cluster Topologies for 6770 Switches

ServerNet clusters that use 6770 switches can have one of three topologies: star, split-star, and tri-star. [Figure 1-20](#), which illustrates only one fabric for simplicity, shows the basic concept of all three of these topologies. Each has its own maximum number of servers that can exist in a cluster: 8, 16, or 24, respectively.

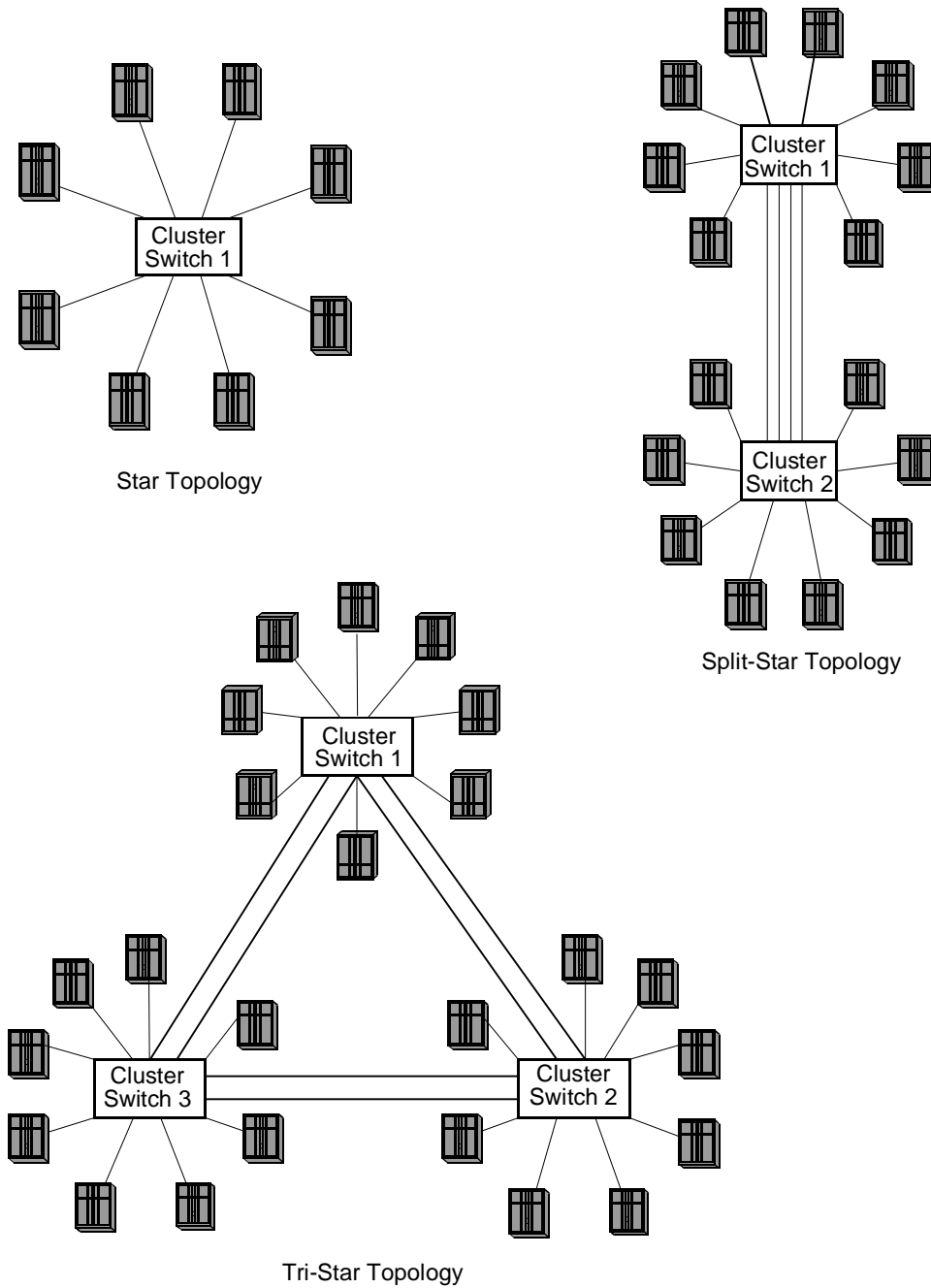
As you look over [Figure 1-20](#), keep in mind that each server icon shown as a component of a cluster is actually a full Tetra 16 multiprocessor server, such as the maximum example shown previously in [Figure 1-19](#) on page 1-37. Each such server, when connected in a cluster, is a **ServerNet node**.

All the servers in a cluster are connected to all other servers in that cluster through fiber-optic cables and cluster switches. The cables and cluster switches constitute the **external ServerNet X and Y fabrics**. This term differentiates these fabrics from fabrics that are contained within a single server, which can be termed the **internal ServerNet X and Y fabrics**.

In all three topologies, each node has two independent connections (the X and Y fabrics) to two independent cluster switches. The loss of a node does not affect the other nodes in the cluster.

You can build your cluster as a subset of either the split-star or the tri-star topology. For example, even though a tri-star topology can contain three cluster switches for each fabric, you can build a tri-star topology that uses only one or two switches for each fabric. By building a subset of a topology, you are positioned to grow the cluster quickly online as your applications grow.

ServerNet clustering is compatible with all other products based on the Expand product. For example, a node in a FOX ring can simultaneously function as a node in a ServerNet cluster. Nodes in a ServerNet cluster can also coexist as systems belonging to an **Expand network**; in that case, each server is both an **Expand node** and a ServerNet node. ServerNet clusters can also coexist with ATM, Ethernet, Fast Ethernet, Token Ring, and other WAN and LAN products.

**Figure 1-20. ServerNet Cluster Star Topologies**

VST080.vsd

# Star Topology

The star topology, introduced with the G06.09 RVU, supports up to eight nodes and requires two 6770 cluster switches—one for the external X fabric and one for the external Y fabric.

---

**Note.** You can configure a single system so that the software necessary for external ServerNet communication is running and ready to communicate with other nodes. However, if no other nodes are connected to the cluster switches, no communication occurs. In this case, communication with remote nodes occurs as soon as the nodes are configured and connected.

---

[Figure 1-21](#) shows both fabrics of an eight-node ServerNet cluster connected in a star topology. As explained previously, each node icon represents a complete multiprocessor server.

ServerNet cables provide ServerNet links between routing devices. In a ServerNet cluster, single-mode fiber-optic (SMF) cables link each node to a cluster switch. The cables are available in 10-meter, 40-meter, and 80-meter lengths. An 80-meter plenum-rated cable is available.

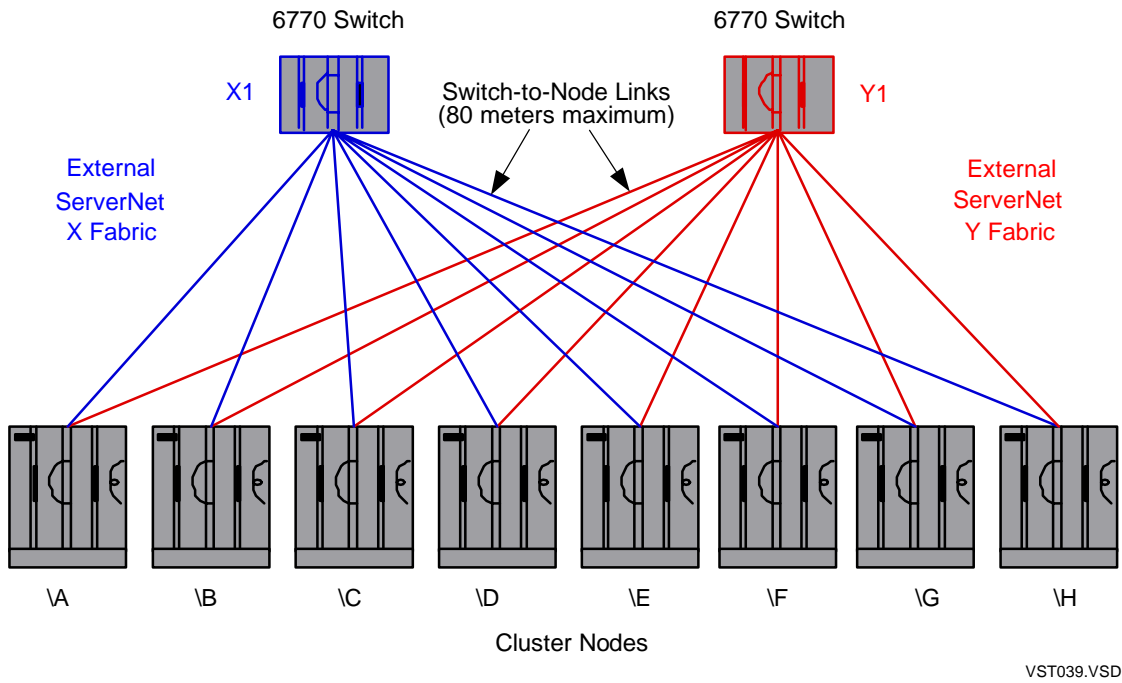
Two ServerNet cables are needed for each node. One cable connects the MSEB in slot 51 of group 01 to the cluster switch serving the X fabric. The other cable connects the MSEB in slot 52 of group 01 to the cluster switch serving the Y fabric.

---

**Note.** Only MSEBs (modular ServerNet expansion boards), not SEBs, are suitable for connecting external fabrics.

---

Figure 1-21. Eight-Node ServerNet Cluster Using Star Topology



# Split-Star Topology

The split-star topology, introduced with the G06.12 RVU, supports up to 16 nodes and is required for clusters using 6770 switches that have more than eight nodes.

[Figure 1-22](#) shows both fabrics of a 16-node ServerNet cluster connected in a split-star topology.

The split-star topology uses up to four cluster switches—two for the X fabric (referred to as X1 and X2) and two for the Y fabric (referred to as Y1 and Y2).

The first cluster switch on a fabric (X1 or Y1) supports ServerNet nodes 1 through 8. The second cluster switch on the same fabric (X2 or Y2) supports ServerNet nodes 9 through 16.

ServerNet clusters using either the split-star topology or the tri-star topology require additional fiber-optic ServerNet cables to connect multiple cluster switches on each fabric. These additional cables are shown in the central area of [Figure 1-22](#) and [Figure 1-23](#).

In the split-star topology, the two cluster switches on each fabric are connected by a four-lane link consisting of four fiber-optic cables. Ports 8 through 11 of the two cluster switches are used for the four-lane link. Traffic travels in both directions across all four lanes of the link.

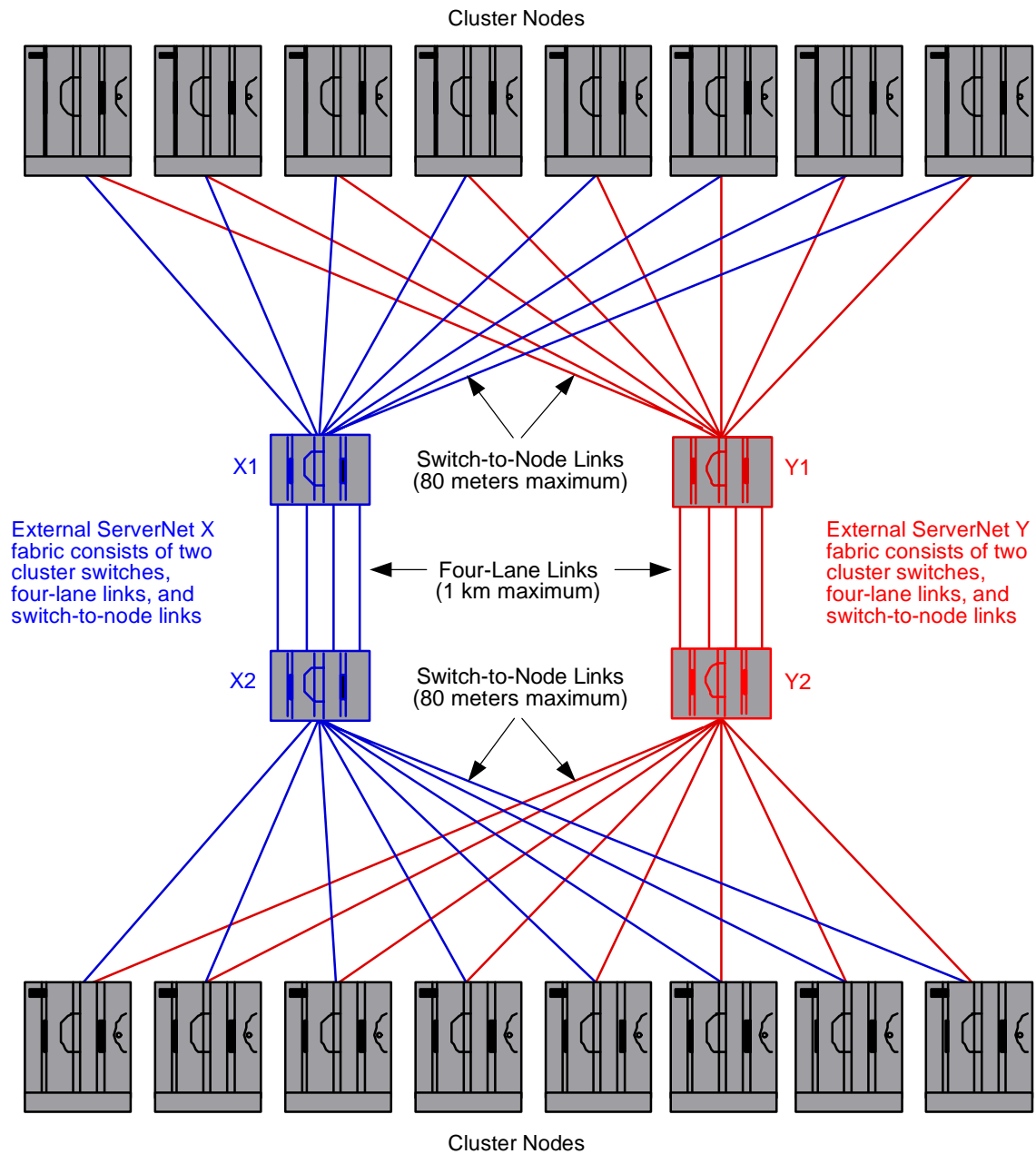
---

**Note.** All four fiber-optic cables are required for a four-lane link, regardless of the number of nodes in the split-star topology.

---

Cluster switches X1 and Y1 send traffic from ServerNet nodes 1 through 8 across the four-lane links to cluster switches X2 and Y2 for routing to ServerNet nodes 9 through 16.

If one lane of a four-lane link is down, traffic between up to four nodes is affected on one of the fabrics. For example, if port 8 is down at either end, ServerNet nodes 1 and 2 cannot communicate with ServerNet nodes 9 through 16 on the affected fabric, and ServerNet nodes 9 and 10 cannot communicate with ServerNet nodes 1 through 8 on the affected fabric.

**Figure 1-22. 16-Node ServerNet Cluster Using Split-Star Topology**

VST069.VSD

# Tri-Star Topology

The tri-star topology, introduced with the G06.14 RVU, supports up to 24 nodes and is required for clusters using 6770 switches that have more than 16 nodes.

[Figure 1-23](#) shows both fabrics of a 24-node ServerNet cluster connected in a tri-star topology.

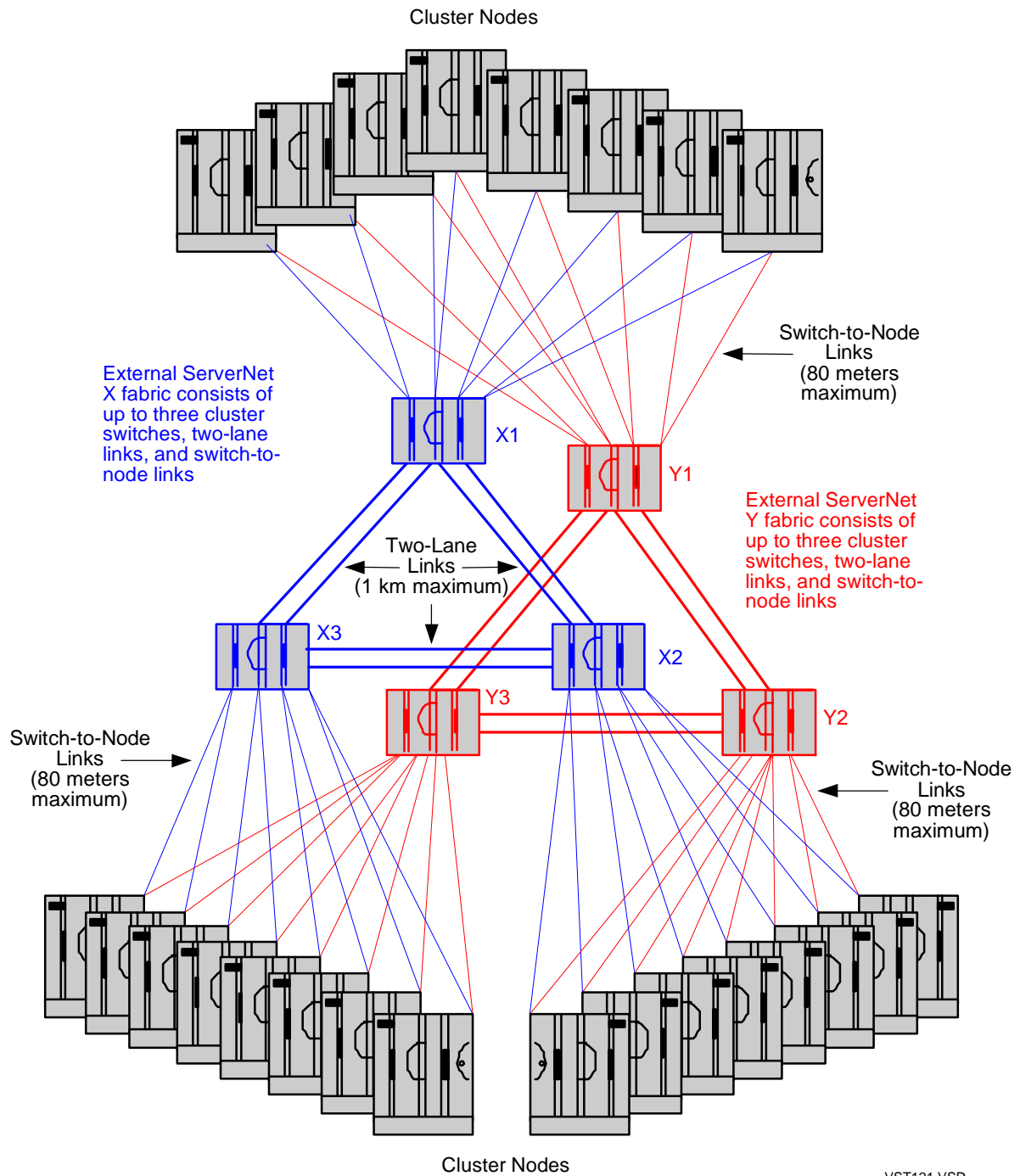
The tri-star topology uses up to six cluster switches—three for the X fabric (referred to as X1, X2 and X3) and three for the Y fabric (referred to as Y1, Y2, and Y3).

The first cluster switch on a fabric (X1 or Y1) supports ServerNet nodes 1 through 8, the second cluster switch on the same fabric (X2 or Y2) supports ServerNet nodes 9 through 16, and the third cluster switch on the same fabric (X3 or Y3) supports ServerNet nodes 17 through 24.

In the tri-star topology, the three cluster switches on each fabric are connected by three two-lane links, each consisting of two fiber-optic cables. Ports 8 through 11 of the cluster switches are used for the two-lane links. Traffic travels in both directions across both lanes of the links.

The tri-star topology features automatic fail-over of ServerNet traffic on the two-lane links. As long as at least one link is operational between cluster switches, processor paths between ServerNet nodes can remain up on both fabrics.



**Figure 1-23. 24-Node ServerNet Cluster Using Tri-Star Topology**

# Layered Topologies for 6780 Switches

ServerNet clusters that use 6780 switches provide more extensive clustering capability than that of the 6770 switches described in the preceding four topics. Instead of being configured in a star topology, the 6780 switches are configured in a layered topology, using one or more **switch layers** and one or more **switch zones**. The number of ServerNet nodes supported depends on the number of switch layers and the number of switch zones. This topic first considers the single-zone case. Multiple zones are considered separately in the succeeding topic, [Connections Between Zones in Layered Topology](#) on page 1-48.

[Figure 1-24](#) illustrates three cases: a single layer, two layers, and four layers. As shown for the first two cases, each layer in the zone supports up to 8 nodes. (The nodes are not shown in the four-layer case for illustrative simplicity.) Therefore, with the maximum number of four layers, 32 nodes is the maximum number of nodes for one zone.

Each pair of 6780 switches (X and Y) forms one cluster switch layer. The layers are numbered from one to four, bottom to top.

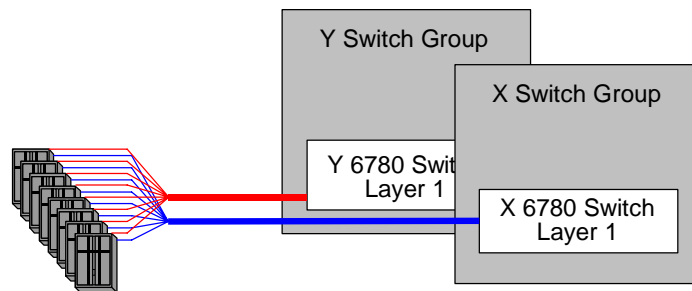
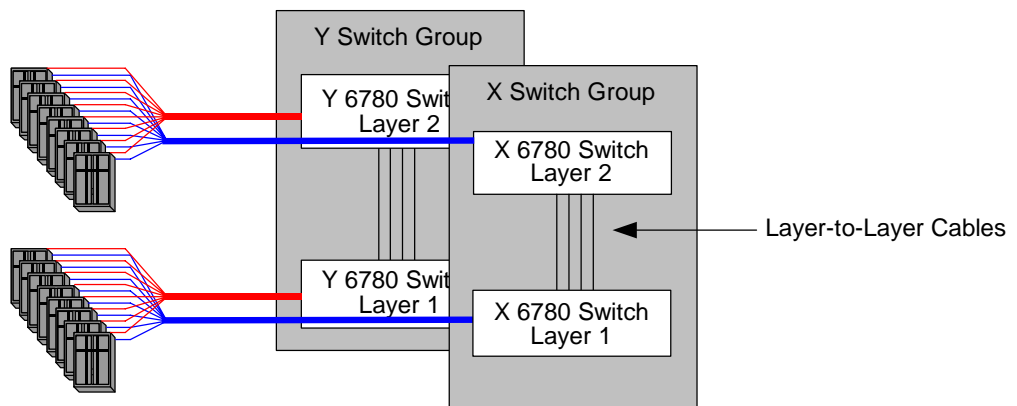
Keep in mind (as in previous illustrations) that a single server icon is actually a ServerNet node consisting of a complete multiprocessor server, such as the maximum example shown in [Figure 1-19](#) on page 1-37.

A **switch group** consists of one to four 6780 switches, and there are distinct X groups and Y groups. Typically, X groups are physically located in one rack, and Y groups in another rack.

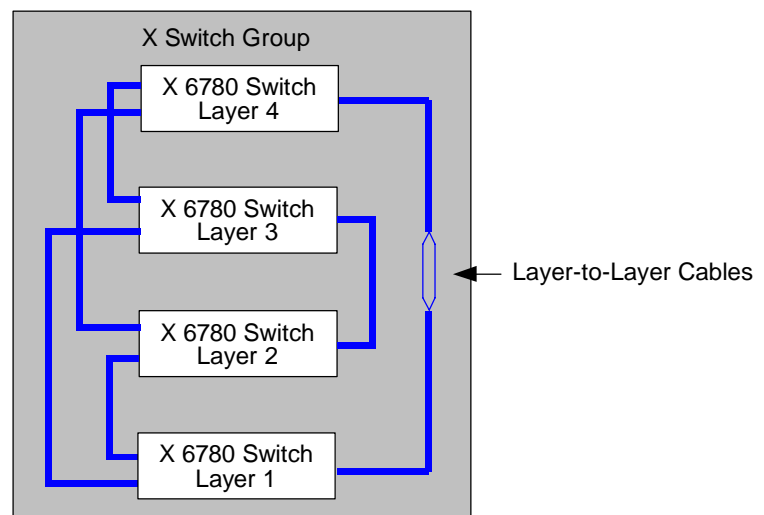
Note that when two or more layers are present, they are interconnected with layer-to-layer fiber-optic cables. Each layer is always connected by four cables to each of the other layers in the group.

In the case of four layers, a tetrahedral arrangement of cables is necessary, so that each layer can directly access any of the other four layers. Because this particular tetrahedron is between layers of switches (rather than between processors), this arrangement is termed a **vertical tetrahedron**. Refer to [Tetrahedral Topology](#) on page 1-26 for a general description of tetrahedral configuration.

In the four-layer case of [Figure 1-24](#), the Y switch group is not shown, but is arranged identically to the X switch group shown.

**Figure 1-24. Single-Zone Cluster Using Layered Topology****Single Layer****Two Layers****Four Layers**

Not Shown for  
Simplicity:  
Nodes and  
Y Switch Group



VST081.vsd

# Connections Between Zones in Layered Topology

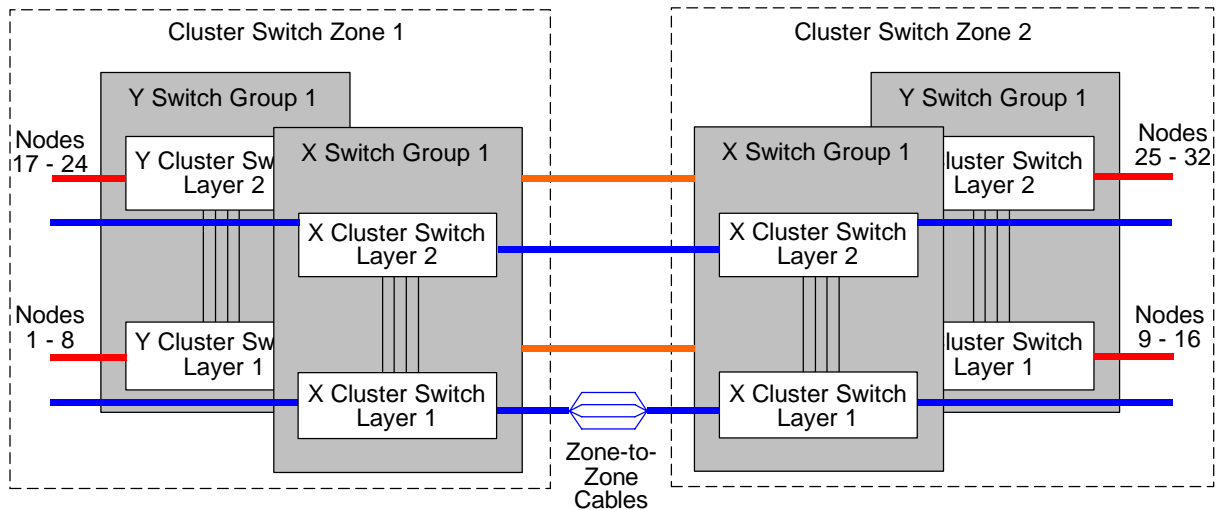
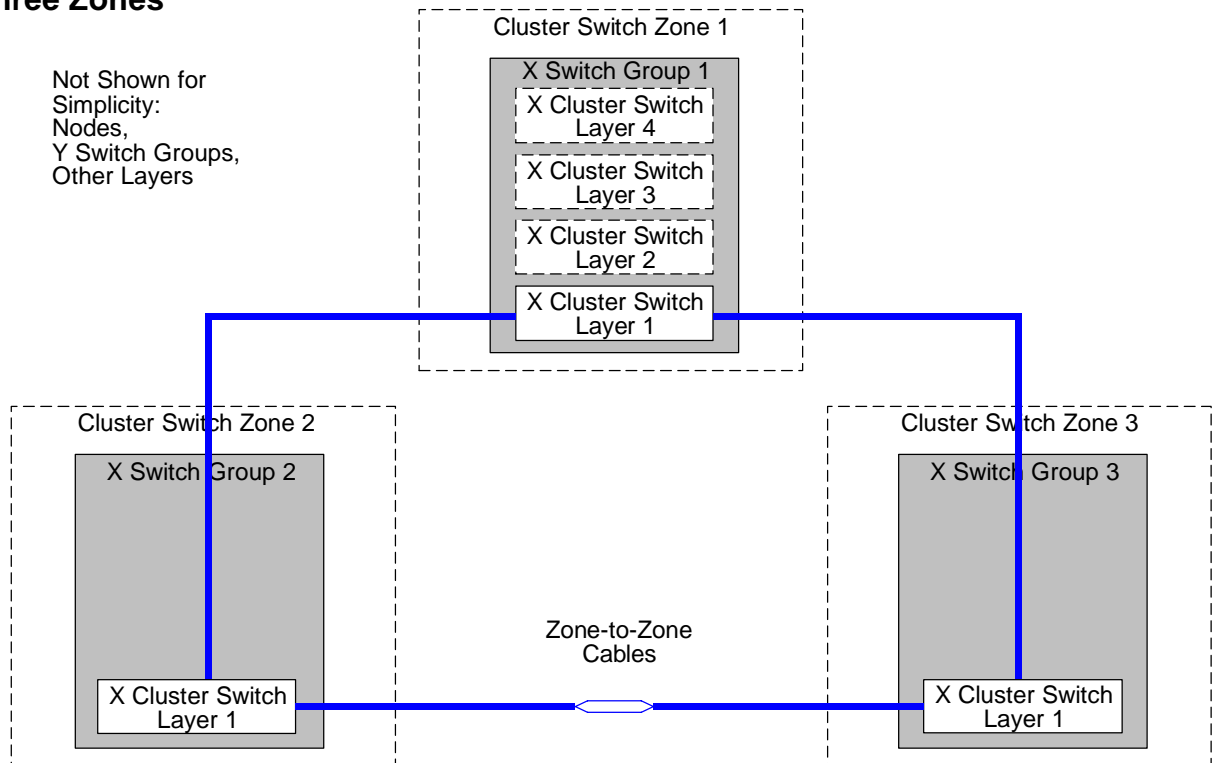
The layered topology described in the preceding topic can be extended to consist of two or three switch zones. [Figure 1-25](#) illustrates these cases.

A switch zone comprises a pair of X and Y cluster switch groups and the ServerNet nodes connected to them. For simplicity, [Figure 1-25](#) does not show the nodes although the two-zone illustration does identify the numbering of the connected nodes. The three-zone illustration includes outline shapes for the four layers in zone 1, but omits them in the other zones. All zones must have exactly the same number of layers.

A maximum of 64 nodes is possible for two zones. For three zones, connections for up to 96 nodes are physically possible, but HP currently supports no more than 64 nodes. HP recommends two zones for best fault tolerance, but three zones can provide ServerNet nodes at three different locations.

Switch zones are interconnected with zone-to-zone fiber-optic cables, and are arranged so that layer 1 switches connect only to layer 1 switches, layer 2 switches to layer 2 switches, and so on. As usual, there are separate X and Y paths.

In the two-zone case, four cables connect each pair of corresponding switches in the two zones. In the three-zone case, two cables connect in a similar manner to each neighboring zone.

**Figure 1-25. Multiple Zones of ServerNet Clusters Using Layered Topology****Two Zones****Three Zones**

VST082.vsd



# 2

# Principles of System Operation

This section describes the fundamental operations that implement the architecture described in [Section 1, Introduction](#). The first few topics define the method of identifying **ServerNet devices** and the structure of **ServerNet packets**, which are the basic entity for communication between ServerNet devices. Following those discussions are several topics describing the way ServerNet transactions are handled by the processor logic. These transaction principles are common to both interprocessor communications and processor-I/O operations. Subsequent sections separately describe the principles that are unique in interprocessor communications and in I/O operations.

The topics covered in this section are:

[ServerNet Devices](#)

[ServerNet Device Identification](#)

[ServerNet Packets](#)

[Packet Transmission and Reception](#)

[Sequence for Outgoing Requests](#)

[Sequence for Responses to Outgoing Requests](#)

[Sequence for Incoming Requests](#)

[Sequence for Responses to Incoming Requests](#)

# ServerNet Devices

[Figure 2-1](#) illustrates the basis for ServerNet device identification, which in turn is the basis for communication between ServerNet devices. A ServerNet device is some kind of ServerNet hardware interface that connects a ServerNet link to a single processor or to an I/O entity that usually controls an array of I/O devices.

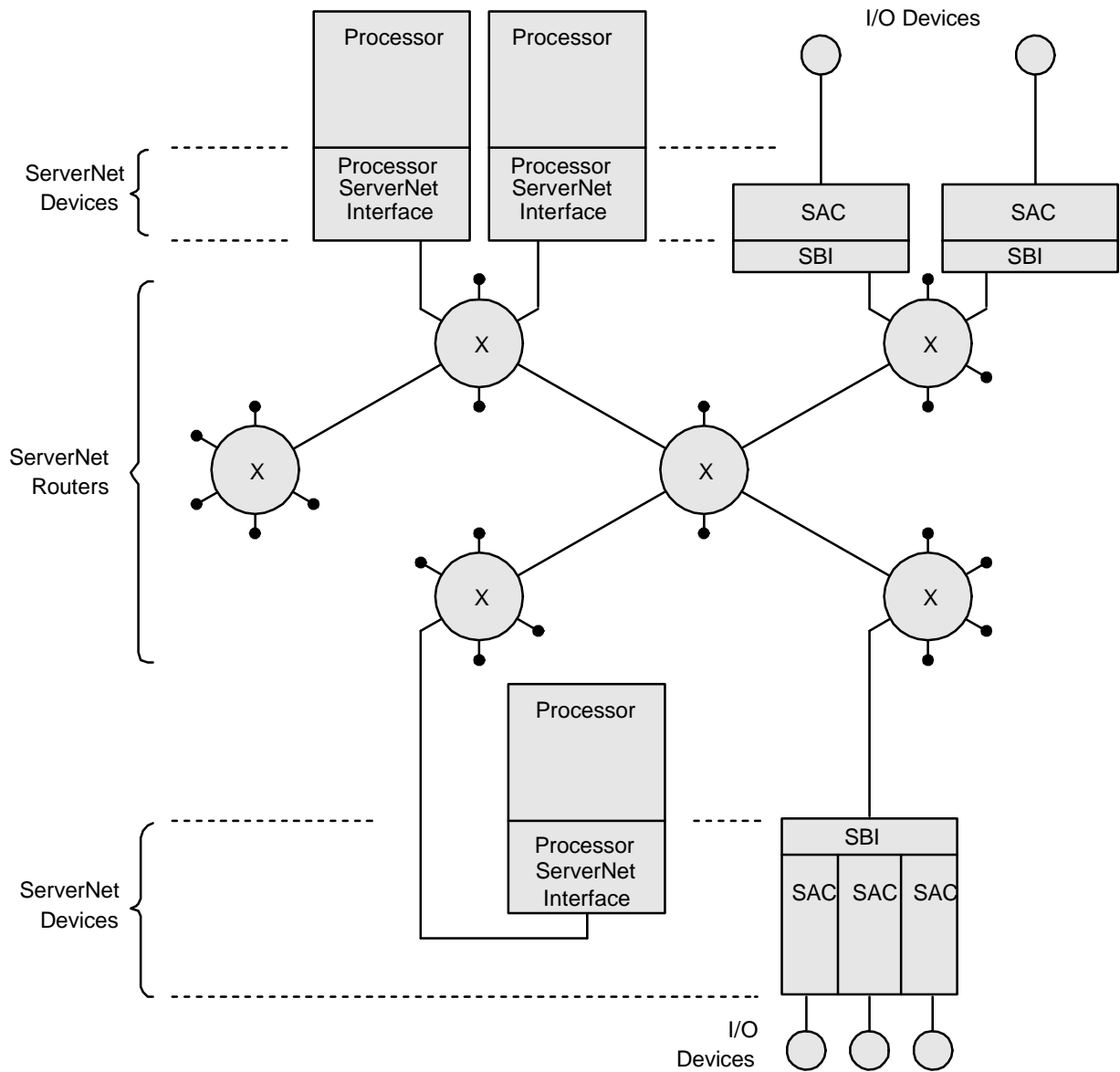
In the case of a processor, that interface is a **processor ServerNet interface**.

In the case of I/O entities, ServerNet addressable controllers (SACs) are always considered to be ServerNet devices. Thus, whenever a single SAC is connected to the ServerNet hardware with its own dedicated ServerNet bus interface (SBI), the SAC is the ServerNet device, not the SBI. Also, I/O devices, whether one per SAC or many per SAC, are never considered to be ServerNet devices.

Sometimes, however, a single SBI can provide the ServerNet interface for several SACs. In that case each SAC is a ServerNet device and, in addition, the SBI can have its own individual ServerNet device identification; it becomes a ServerNet device, just like a SAC. Note in the example in the lower right corner of [Figure 2-1](#) that there is only one link to the ServerNet routers, even though in this case there are four ServerNet devices.

Note the distinction that ServerNet routers (in the middle of the diagram) are pass-through switching circuits and are never referred to as ServerNet devices. To emphasize this distinction, a true ServerNet device is sometimes termed a **ServerNet end device**.



**Figure 2-1. A ServerNet Device Can Be a Processor, Controller, or Bus Interface**

SBI = ServerNet Bus Interface  
 SAC = ServerNet Addressable Controller

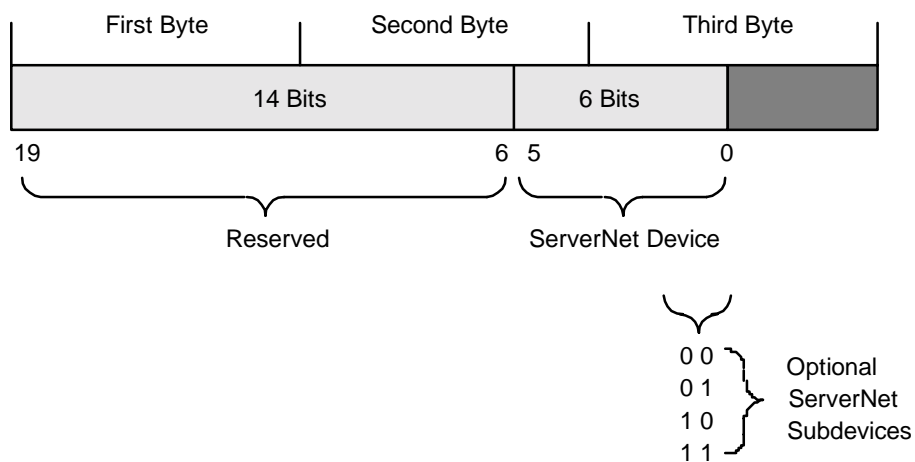
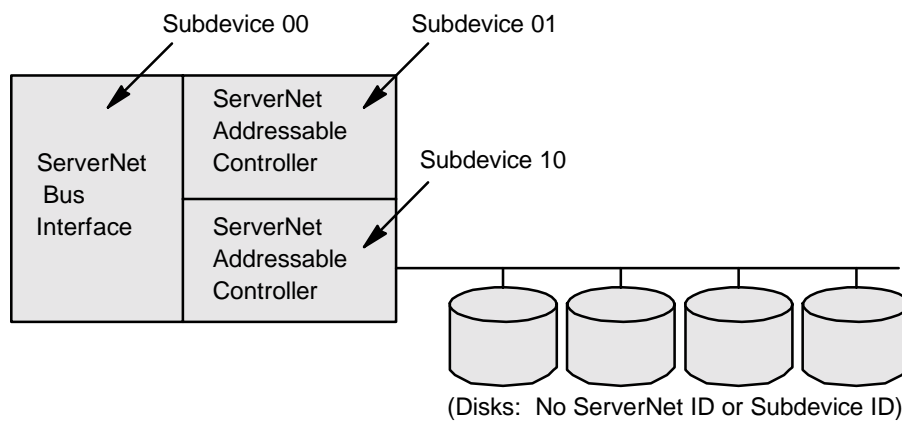
VST330.vsd

## ServerNet Device Identification

When ServerNet devices communicate with each other, they provide a **ServerNet identification (ID)** for both the target and the originator of the transmission. The format of the ServerNet ID is shown in [Figure 2-2](#).

For current purposes, the entire 20-bit value can be considered as an undivided ServerNet ID. The reserved bits are not currently implemented. Also, the term “ServerNet device” does not refer to I/O devices; as stated in the preceding topic, I/O devices are never ServerNet devices.

In cases where multiple SACs interface with one SBI (see [Figure 2-1](#)), low-order bits in the ServerNet ID can be used to distinguish each one as a “subdevice.” The example in the upper part of [Figure 2-2](#) shows two low-order bits being used as the ServerNet subdevice field. The lower part of the figure shows how this encoding could be interpreted for hardware references.

**Figure 2-2. ServerNet ID Identifies ServerNet Devices and Subdevices****ServerNet ID****Use of Subdevice ID**

VST331.vsd

# ServerNet Packets

Communications between ServerNet devices at a fundamental level occurs in the form of variable-length **ServerNet packets**. Generally, packet transmissions occur in pairs called a **ServerNet transaction**. The device originating the transaction sends a **request packet** and the target device sends back a **response packet**. (exceptions to this one-for-one arrangement can happen for diagnostic and status transactions.

There are two basic kinds of transactions: read transactions and write transactions. Because each transaction requires a request and a response, there are four possible types of packets, as shown in [Figure 2-3](#). The structure of each packet depends on whether it is a read request, a read response, a write request, or a write response.

Each packet begins with a fixed-length (8-byte) header that contains the information shown in the lower part of [Figure 2-3](#). Notice that the first information to be transmitted on a ServerNet link is the 20-bit destination ServerNet ID (format shown in [Figure 2-2](#)). Having this ServerNet device ID at the start of a packet permits the routers to start sending the packet out through the appropriate port as soon as the first three bytes arrive at the router.

Additional information in the header includes the source ServerNet ID, which the target device will use to identify the sender of the packet. The 4-bit transaction type, in combination with the 1-bit request/response identifier, specifies which packet type is coming.

The 8-bit data length specifier architecturally allows specifying a data field (D) much larger than current hardware implementations allow, which is 64 bytes.

The 4-bit transaction ID permits a ServerNet device to originate and have outstanding up to 16 transactions before receiving a response for any of them. As each response arrives, it can be associated with corresponding earlier requests by using this transaction ID value. Although this field allows for 16 outstanding transactions, individual end device designs might not use this full capability. The primary/alternate path bit specifies the preferred fabric for transmission, X or Y.

The acknowledge/negative acknowledge bit is applicable only in response packets. A positive acknowledge specifies that the corresponding request packet was successfully received and passed on to other logic in the receiving ServerNet device. A negative acknowledge specifies that, although the packet was received, there was some problem, such as an invalid ServerNet address.

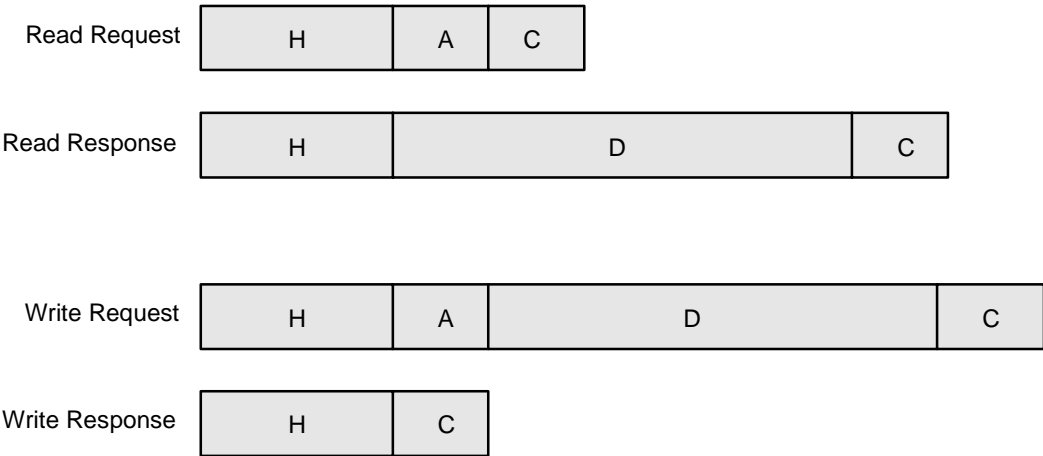
The 4-byte **ServerNet address** can be thought of as a virtual memory address within the receiving device, as it most commonly actually is—even for I/O ServerNet devices. In some cases, however, this address could specify a register or some specific interrupt. The address is needed only for requests, to specify where in the target device to read or write data. It is not needed in response packets.

The variable-length data field comes in the read response, after having read the data from the specified address in the target device, and in the write request, along with the address telling where to write the data in the target device.

The CRC code checks for errors in all preceding fields, however many there are.

**Figure 2-3. A ServerNet Transaction Has Both a Request and a Response Packet**

**ServerNet Packet Types**



**ServerNet Packet Layout**

- H    8-Byte Header
  - 20-Bit Destination ServerNet ID (DID)
  - 20-Bit Source ServerNet ID (SID)
  - 4-Bit Transaction Type
  - 8-Bit Data Length
  - 4-Bit Transaction ID
  - 1-Bit Primary/Alternate Path
  - 1-Bit Acknowledge/Negative Acknowledge
  - 1-Bit Request/Response
  - 5 Bits Reserved
  
- A    4-Byte ServerNet Address
  
- D    0-Byte to 64-Byte Data
  
- C    4-Byte Cyclic Redundancy Check (CRC) Code
  - on preceding fields: H, (A), (D)

VST332.vsd

# Packet Transmission and Reception

ServerNet devices take different actions and use different logic based on whether the request part of a transaction originated within the local ServerNet device or in some other ServerNet device (called “remote ServerNet device” in [Figure 2-4](#)). The reason for this difference is that the local device has all the necessary information about any transaction that it originated, whereas an incoming request from a remote device is basically unexpected and carries its own information. [Figure 2-4](#) separately illustrates the handling of both locally initiated and remotely initiated requests, for all four types of packets.

In this figure, the previously shown “processor ServerNet interface” is broken down into two major components, labeled BTE and AVT. (There are other components, but these are the two of interest here.)

BTE is **block transfer engine**, and it is used for all locally initiated transaction handling, whether read or write, request or response. In the read case, the BTE starts the transaction by sending a read request to the remote device, including the address for reading within that remote device. The remote device must respond with a read response, including data of up to 64 bytes. Because the BTE tells the remote device where to read the data and where to send it, the local processor’s BTE “pulls” the data from the remote device.

In the write case, the BTE starts the transaction by sending a write request to the remote device, including the address for writing within that remote device, as well as up to 64 bytes of data to write. After the remote device has written the data, it sends back a simple write response (header and CRC only). Because the BTE tells the remote device where to write the data and actually provides the data, the local processor’s BTE “pushes” the data to the remote device.

AVT is **access validation and translation**, and it is used for all remotely initiated transaction handling. The remote device could be another processor (in which case it would have used its BTE to initiate the transaction) or it could be, for example, a ServerNet addressable controller (SAC).

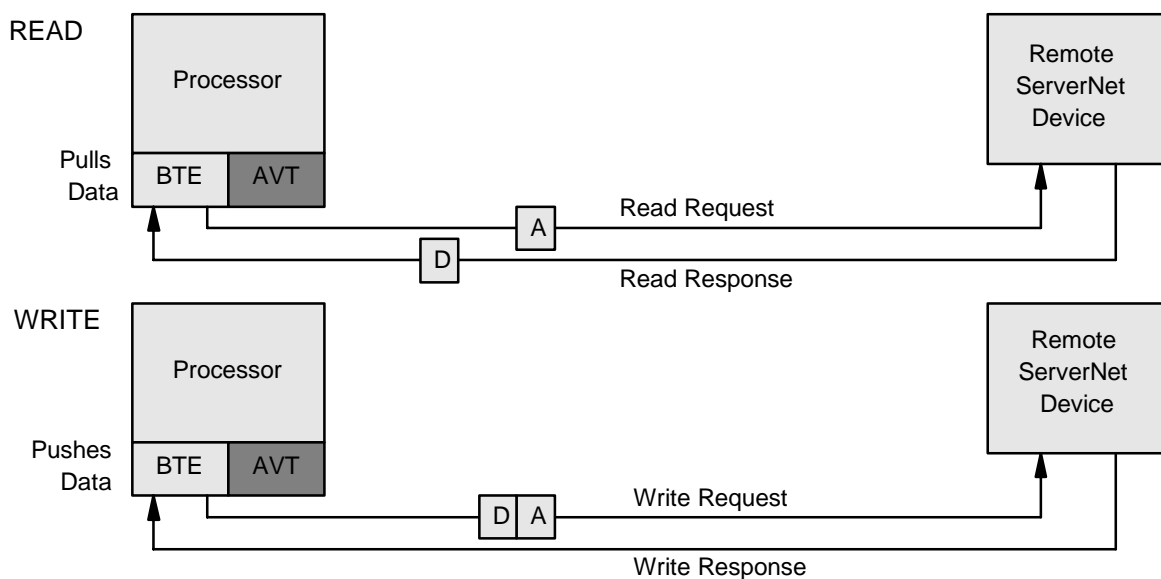
In the read case, the AVT receives a read request along with an address for reading the requested data. Because this is an unexpected request, the AVT first must do an “access validation” to determine whether the particular remote device is permitted to access the portion of memory it wants to read. If the access is permitted, the AVT next translates the supplied virtual address to a physical address, gets the requested data, puts it into a read response packet, and sends the packet to the remote device. In this case, the remote device is pulling the data.

In the write case, the AVT receives a write request along with an address and up to 64 bytes of data that the remote device wants written into the local memory. The AVT first validates the supplied address, then proceeds if the access is permitted. The AVT translates the virtual address to physical and stores the data in the processor memory. Finally it sends a write response back to the remote device. In this case, the remote device is pushing the data.

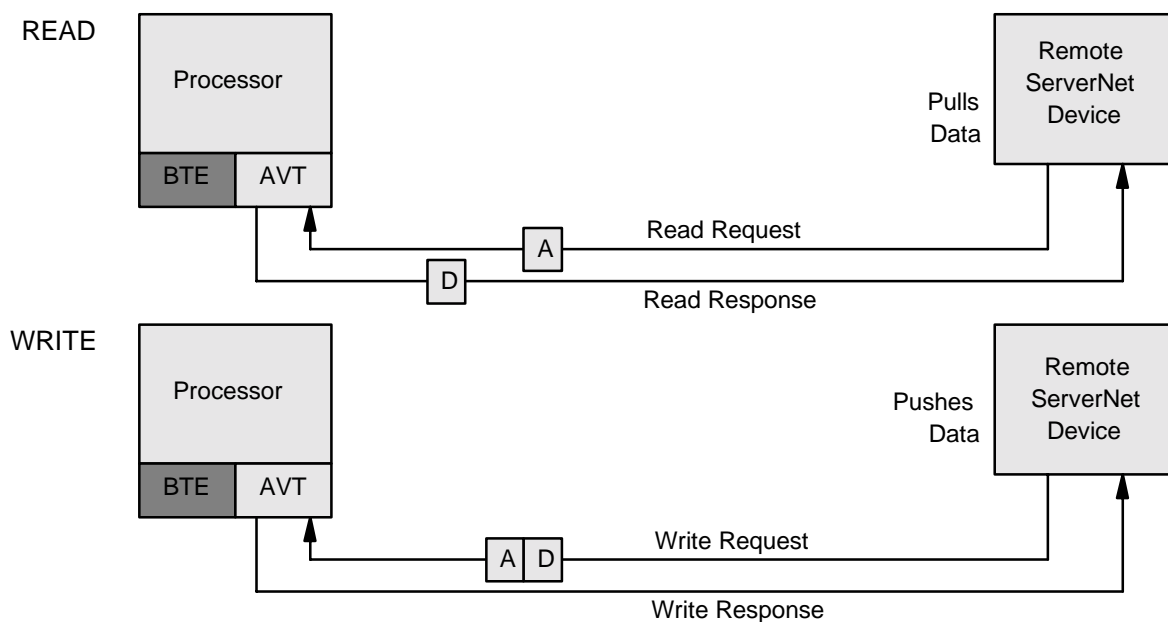
The BTE and AVT both have direct-memory access (DMA) engines and perform their read and write operations without processor intervention, using buffer addresses that, by prior agreement, have been allocated individually to particular ServerNet devices.

**Figure 2-4. BTE Handles Local Requests; AVT Handles Incoming Requests**

**Locally Initiated Transaction Handling**



**Remotely Initiated Transaction Handling**



VST333.vsd

## Sequence for Outgoing Requests

This topic and the next one describe in more detail the BTE operations outlined in the preceding topic. (Then the final two topics describe the AVT.)

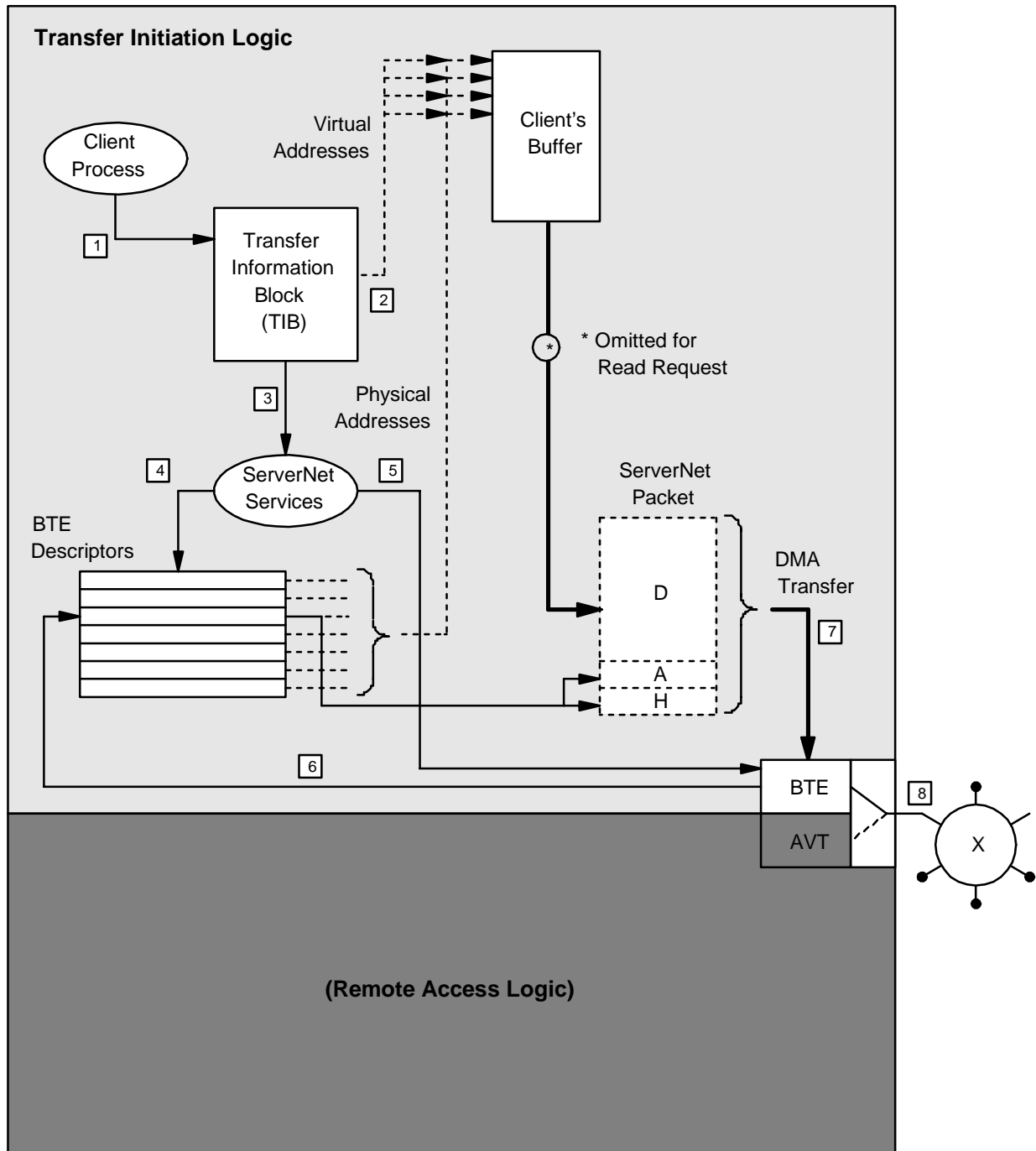
[Figure 2-5](#) illustrates the sequence of events for locally initiated read and write requests. The only difference for the read and write sequences is that data (D) is not included in read requests. The following steps correspond to the numbered callouts in the figure.

1. A client process (such as an I/O process or the message system) initiates a request by first building a **transfer information block** (TIB) in memory. A transfer information block consists of a header and one or more buffer descriptor structures.
2. Once the TIB is completed, it defines the virtual addresses for the buffer that is to be the source of write data or the destination of incoming read data.
3. The client process now invokes **ServerNet services**, supplying the location of the TIB.
4. The ServerNet services, using information from the TIB, builds a linked series of **BTE descriptors** for the use of the block transfer engine. These descriptors contain the (translated) physical addresses of the client's buffer (one descriptor for one page), along with other necessary transfer information.
5. ServerNet services now passes control to the hardware of the BTE engine, supplying the location of the BTE descriptors.
6. The BTE hardware gets information from the BTE descriptors to build the header of the outgoing packet (H) and the address in the target ServerNet device (A) that is applicable for reading or writing the data.
7. The BTE hardware completes the outgoing packet. In the case of a write request, the BTE hardware uses DMA and the supplied physical addresses to include data (D) in the packet.
8. The BTE hardware sends the completed packet through the processor ServerNet interface to the connected port of the processor's router, thus beginning transmission to the target ServerNet device.

At this point, the BTE hardware updates internal registers for byte count and address, and then proceeds to the next packet until the byte count is exhausted.



**Figure 2-5. Transfer Initiation Logic Uses BTE Descriptors for Outgoing Requests**



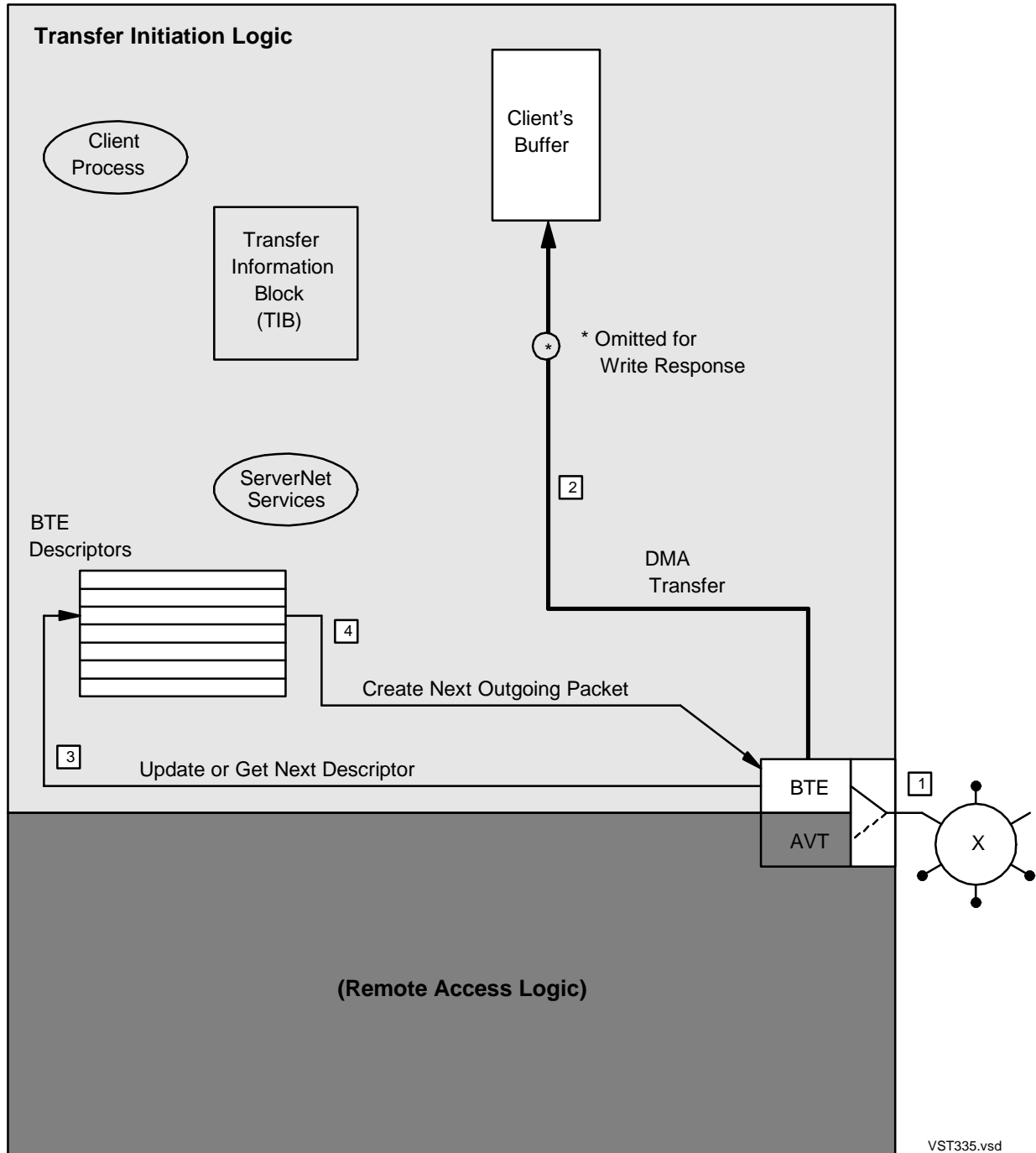
VST334.vsd

## Sequence for Responses to Outgoing Requests

After the target ServerNet device has received and taken action on the request described in the preceding topic, it sends a response packet back to the sending ServerNet device. [Figure 2-6](#) illustrates the actions taken when the response packet is received back at the requesting device.

1. The incoming response packet is received by the processor ServerNet interface. Upon examining the header of the packet, the processor ServerNet interface determines that this is a response packet and therefore passes control to the BTE hardware. (Only the BTE logic would be interested in incoming response packets.)
2. If the original request was a read request, the response packet contains data. The BTE hardware, in this case, does a DMA transfer to memory, using the physical address available in the current BTE descriptor.
3. If a page crossing is needed to complete the read or write request, the BTE hardware advances to the next linked BTE descriptor for additional information. (See Step 4 in the preceding topic.)
4. The BTE hardware obtains information needed to create the next packet and transmits it to the target device through the ServerNet hardware. (Steps 7 and 8 in the preceding topic.)

Steps 3 and 4 can be repeated several times in the case of a lengthy request, until the end of the BTE descriptor list is reached.

**Figure 2-6. BTE Hardware Handles Responses to Outgoing Requests**

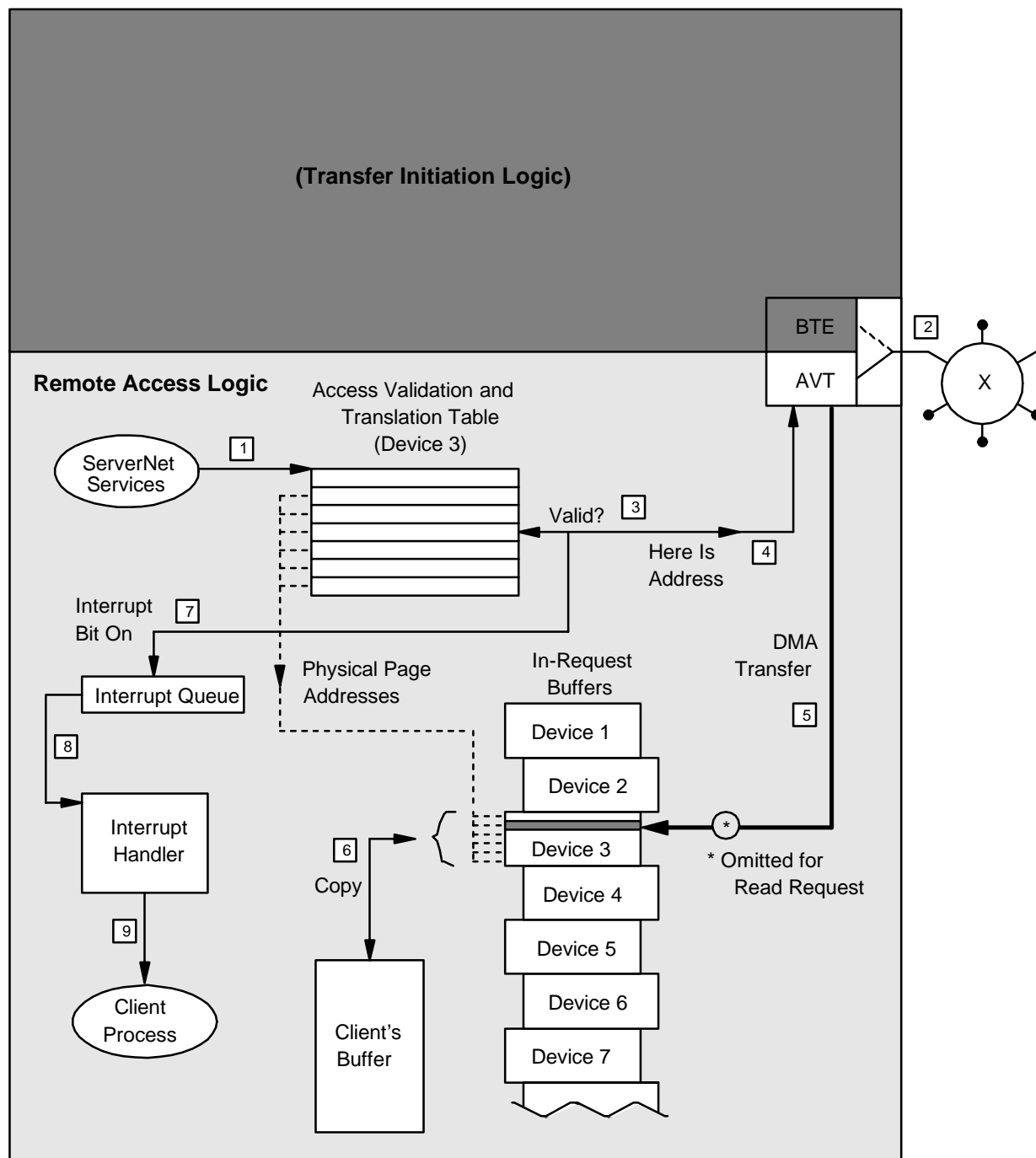
## Sequence for Incoming Requests

[Figure 2-7](#) illustrates the sequence of operations for an incoming request, one that did not originate in this ServerNet device. In this case, the remote access logic is used instead of the transfer initiation logic (shaded out) that was used in [Figure 2-5](#) and [Figure 2-6](#).

1. Prior to being able to handle incoming requests, an access validation and translation table must have been set up, as well as buffer space for each ServerNet device that can possibly make requests to this (illustrated) device. The example shown arbitrarily assumes that “device 3” is sending the request. Thus ServerNet services will have set up the access validation and translation table (AVTT) for that device. Entries in the AVTT contain information for virtual-to-physical address translation.
2. The incoming request packet is received by the processor ServerNet interface. Upon examining the header of the packet, the processor ServerNet interface determines that this is a request packet. It is unexpected in the sense that this device (specifically the BTE logic) did not originate this transaction. The processor ServerNet interface therefore passes control to the access validation and translation (AVT) hardware.
3. The AVT hardware accesses the AVTT to determine whether the address supplied in the header of the packet is valid—that is, that it actually addresses some location in the buffer space allocated for “device 3.” Virtual addresses are being compared at this time.
4. If the access is valid, the AVT hardware reads the table entry information and translates the address supplied by the packet into a physical address. The AVT hardware at this point also returns a response packet (see next topic). If the access is not valid, the response packet includes a negative acknowledge in the header.
5. If the incoming valid request is a write request, the AVT hardware writes the packet data into the addressed locations within whatever buffer space allocated for “device 3.” The addressed locations may be an in-request buffer, as shown, or in some cases can be the client’s buffer.
6. If a dedicated in-buffer is used, the client process periodically must copy the data to its own (or some other) buffer.
7. If the incoming request address is valid but addresses a particular location that has been agreed upon as an interrupt location (the AVTT entry at that location has its interrupt bit set on), the packet is placed in an interrupt queue. A typical reason for doing this might be to signify that all packets for the current request have been exchanged.
8. In queue order, the packet is examined by exception-handler millicode, and the appropriate interrupt is passed up to a higher-level interrupt handler.

9. The interrupt handler awakens the client process to the fact that an interrupt has occurred and the client process needs to take appropriate action.

**Figure 2-7. Remote Access Logic Uses AVT Table for Managing Incoming Requests**

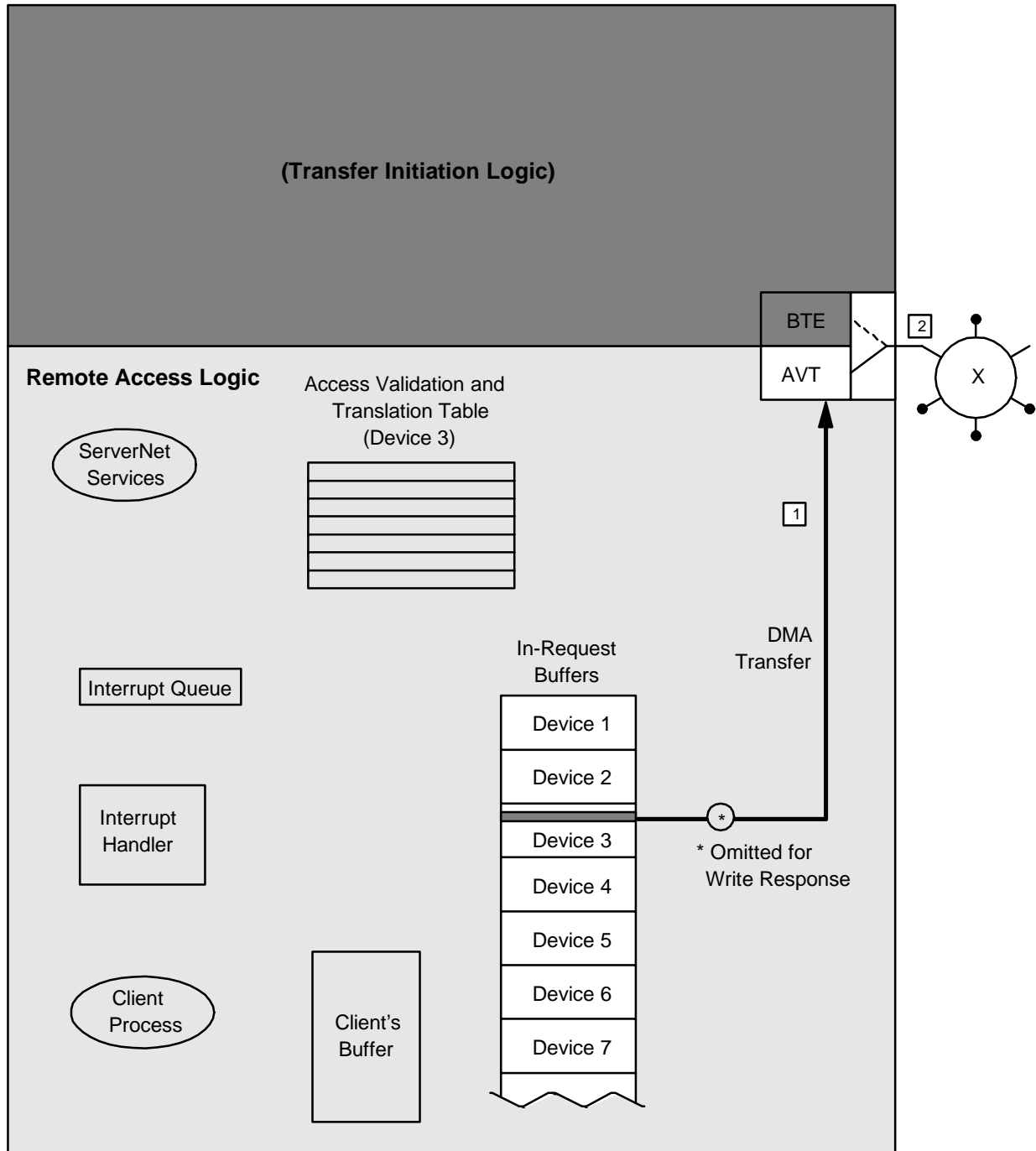


VST336.vsd

## Sequence for Responses to Incoming Requests

[Figure 2-8](#) shows the sequence for sending a response back to the sending ServerNet device after this ServerNet device has received and taken action on the received request. In the case of a write request, the action would have been to write the data to a buffer, as described in the preceding topic. In the case of a read request, the action is part of the response, as described here.

1. If the incoming request was a read request, the AVT hardware reads from the device's buffer up to 64 bytes to include in a read response packet.
2. The AVT hardware creates the read or write response packet and, through the processor ServerNet interface, sends the packet to the requesting device through the ServerNet hardware.

**Figure 2-8. AVT Hardware Handles Responses to Incoming Requests**

VST337.vsd





# TNS Data Formats and Number Representations

This section describes heritage information relating to the compatability issues involved in maintaining compatability from the earliest Tandem (now HP) CISC-based systems to the current RISC-based systems. The distinction between these two major types of systems is usually denoted by the descriptors TNS and TNS/R.

The topics in this section are:

[TNS Words and RISC Words](#)

[TNS Data Formats](#)

[TNS Byte Instructions](#)

[Instructions for Unsigned and Signed Arithmetic](#)

[Instructions for Decimal and Floating-Point Arithmetic](#)

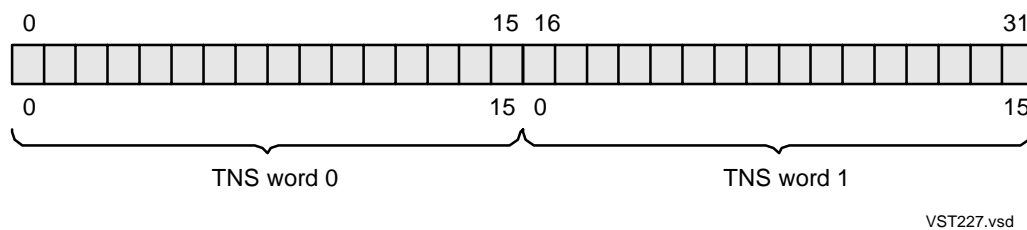
# TNS Words and RISC Words

A RISC word is four bytes (32 bits) wide and always begins at a four-byte memory boundary. Native execution mode and most hardware subsystems in the NonStop S-series servers use this notion of word size and word alignment.

However, the TNS instruction set, as originally used in the earliest NonStop systems, is defined in terms of a two-byte (16-bit) word size. Upward compatibility with that original instruction set is preserved in all NonStop servers, including the NonStop S-series servers.

As shown in [Figure 3-1](#), two TNS words fit into each RISC word. The TNS word is two bytes wide and is placed in memory at a two-byte memory boundary. Any two consecutive TNS words, aligned on any two-byte boundary, constitute a TNS doubleword. Any four consecutive TNS words, aligned on any two-byte boundary, constitute a TNS quadrupleword.

**Figure 3-1. Two TNS Words Are Equivalent to One RISC Word**



To provide a distinction for the TNS definition of a word, references to a **TNS word** are generally preceded by the TNS modifier where written or verbal ambiguity could occur. Because this manual necessarily deals frequently with TNS instructions, such as in [Section 11, TNS Instruction Set](#), TNS terminology is used extensively and reader comprehension is assumed.

In TNS or accelerated TNS programs, all nonbyte data must be placed on two-byte memory boundaries to get consistent, correct results from TNS instructions. TNS compilers automatically take care of this requirement for all normally declared data. The application programmer must take care to follow this rule when doing dynamic allocation of nonbyte data through address arithmetic and pointer conversions, or when navigating a buffer containing nonbyte data. For details, see the data misalignment information that is provided in the various language reference manuals.

TNS instructions using an extended address to access nonbyte data, such as LWX, all require the extended byte address to be even. The behavior of the instruction is *undefined* if the given address is odd. The instruction could behave in one of three ways:

- raising an instruction failure trap
- implicitly rounding down the address to the next lower even address

- completing the operation without rounding down the address

On the original TNS CISC processors (NonStop Cyclone and earlier), invalid odd addresses always give the second behavior. On TNS/R processors, including the NonStop S-series servers, the TNS and accelerated TNS execution modes give a mix of these three behaviors for odd addresses. The TNS application programmer should not count on any consistency, but should instead avoid forcing odd addresses into extended pointers for nonbyte data.

Beginning with release version update G06.17, the operating system generates EMS error events when invalid odd addresses are used in TNS or accelerated TNS modes. Further, the TNS emulators can be configured to replace former round-down cases by instruction-failure traps or by completion without round-down. This is controlled by new Subsystem Control Facility (SCF) commands. The choice applies to all TNS processes on that system. For details see the data misalignment information that is provided in the *Accelerator Manual*, and in the *SCF Reference Manual for the Kernel Subsystem*.

## Endian Convention

In keeping with original conventions and **big-endian** design, bit numbering within an object is left-to-right, the high-order, leftmost bit being bit 0. Manuals that specifically describe the RISC chips generally assume the reverse. However, that difference is only a documentary convention; no TNS or RISC instructions numerically designate bit positions within a word.

## Other Conventions

In this manual, a number surrounded by brackets is used to denote an individual element (that is, byte, word, TNS doubleword, or TNS quadrupleword) in an array or block of elements. For example, to indicate the fourth element in a word block (beginning with element 0), the following notation is used:

WORD [ 3 ]

When referring to a block of words (or any elements), the first element is indicated by the element number that is the lowest numerically; the last element has the highest element number. The first and last elements are separated by a colon. For example, to indicate the second through twentieth words in a block, the following notation is used:

WORD [ 1 : 19 ]

The following notation is used in this manual (and in the TAL language) to describe bit fields:

WORD . < 4 : 15 >

This example defines a field within a word starting with bit 4 and extending through bit 15. To indicate just bit 0, the following notation is used:

WORD . < 0 >

# TNS Data Formats

The available TNS data element sizes are single bytes, two-byte words, four-byte doublewords, and eight-byte quadruplewords. Signed integer data formats are available as word, doubleword, and quadrupleword elements. Floating-point data formats are available as doubleword and quadrupleword elements. See [Figure 3-2](#).

The instruction set of all NonStop processors supports individual access to and operations on 8-bit **bytes**, 16-bit TNS words, 32-bit TNS doublewords, and 64-bit TNS quadruplewords. The figure on the facing page shows all of these data formats, plus the legacy TNS **floating-point** and **extended floating-point** data formats.

A 16-bit **TNS word** can be a signed integer, an unsigned integer, a short pointer for addressing within a particular memory segment, or a TNS instruction. As indicated in the [Figure 3-2](#), a TNS word is equivalent to two bytes. In TNS or accelerated execution modes, all TNS word operands must begin at two-byte memory boundaries for TNS instructions to operate correctly. In native execution mode, 16-bit operands can begin at any byte, but there is a large performance penalty if the operand is not aligned to a two-byte memory boundary.

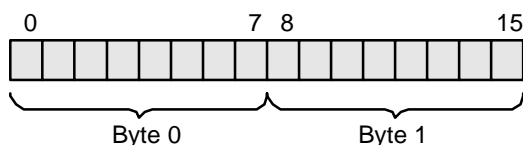
A TNS **doubleword** can be a 32-bit signed integer, a 32-bit extended address, a single-precision floating-point number, or a RISC instruction. A TNS doubleword is equivalent to four bytes. In TNS or accelerated execution modes, all TNS doubleword operands must begin at a two-byte memory boundary for correct execution, and there is no benefit to four-byte alignment. In native execution mode, 32-bit operands can begin at any byte, but there is a large performance penalty if the operand is not fully aligned to a four-byte memory boundary.

A TNS **quadrupleword** can be a 64-bit signed integer, a fixed-point number represented as a 64-bit signed integer with an assumed scale factor (radix point) fixed at compile time, or an extended-precision TNS floating-point number. A quadrupleword is equivalent to four TNS words or eight bytes. In TNS or accelerated execution modes, all TNS quadrupleword operands must begin at a two-byte memory boundary for correct execution, and there is no benefit to four-byte or eight-byte alignment. In native execution mode, 64-bit operands can begin at any byte, but there is a large performance penalty if the operand is not fully aligned to an eight-byte boundary.

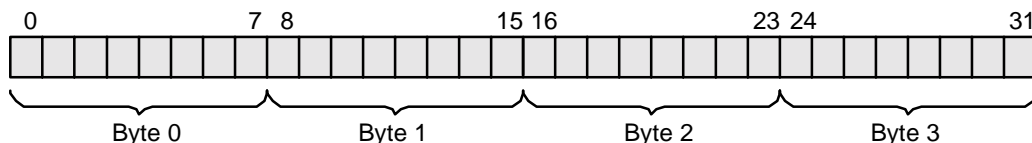
The single-precision and extended-precision floating-point formats are shown here for comparison and are discussed in [Instructions for Decimal and Floating-Point Arithmetic](#) on page 3-10.

**Figure 3-2. A Variety of TNS Data Formats Are Available**

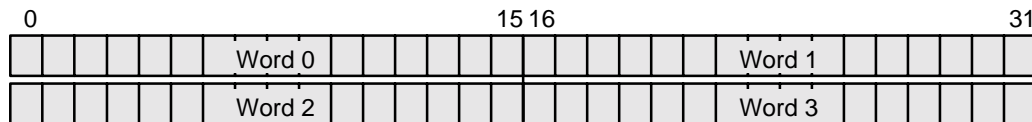
A TNS word can contain two bytes:



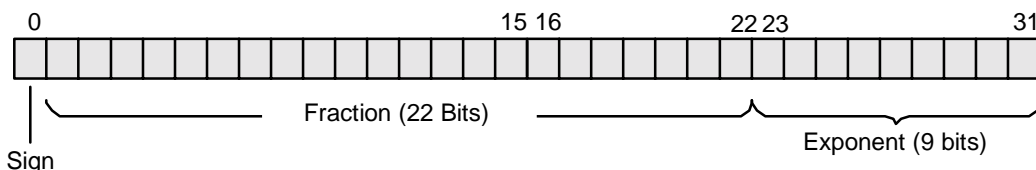
A TNS doubleword can contain four bytes:



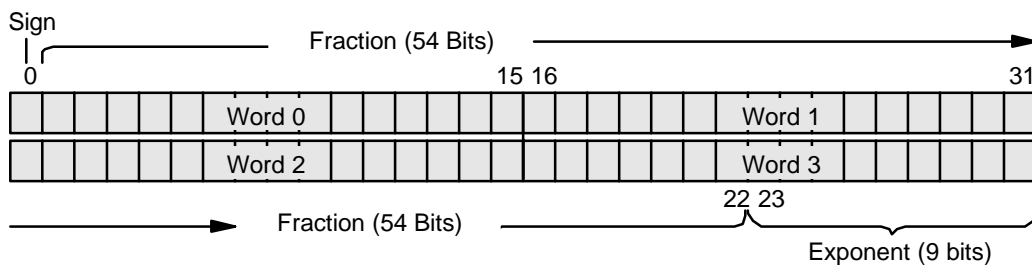
Four TNS words form a TNS quadrupleword:



Two TNS words are needed to form a TNS-format floating-point doubleword:



Four TNS words are needed to form a TNS-format extended floating-point quadrupleword:



VST228.vsd

# TNS Byte Instructions

Two bytes can be stored in a TNS 16-bit word. Bytes within a multibyte object or data area are addressed and numbered according to the big-endian convention—that is, the numbering is from left to right. Within a 16-bit word, the most significant byte occupies WORD.<0:7> (the left half); the least significant byte occupies WORD.<8:15>. This arrangement is shown in [Figure 3-3](#).

Programs that assume the little-endian convention (right-to-left numbering) must be adjusted when being converted for TNS or TNS/R processing.

Various instructions are provided in the TNS instruction set for manipulating byte operands, either individually or as blocks of bytes. For example, bytes can be transferred between the register stack and a memory location (LDB, STB, LBP, LBA, SBA, LBX, SBX, LBXX, SBXX). Blocks of bytes can be moved, compared, or scanned (MOVB, COMB, SBW, SBU, MVBX, MBXR, MBXX). Also, strings of bytes using ASCII-coded digits can be converted to and from their corresponding quadrupleword integer values (CAQ, CAQV, CQA).

The BTST (Byte Test) instruction is provided for testing the character class (that is, ASCII alphabetic, numeric, or special) of a byte operand. This test is based on a special interpretation of the condition code. This same special condition code is also set by all byte-load instructions.

Note that byte-store operations all store the rightmost eight bits of a 16-bit register with no overflow detection.

No instructions are provided for direct support of signed bytes, nor for arithmetic on unsigned bytes. Arithmetic may be performed on unsigned bytes indirectly by using a byte-load instruction (which zero-extends to 16 bits) and then using a 16-bit arithmetic operation. The result, in this case, can be outside the range 0 through 255, and any overflow detection is based on the 16-bit range, not on the 8-bit range.

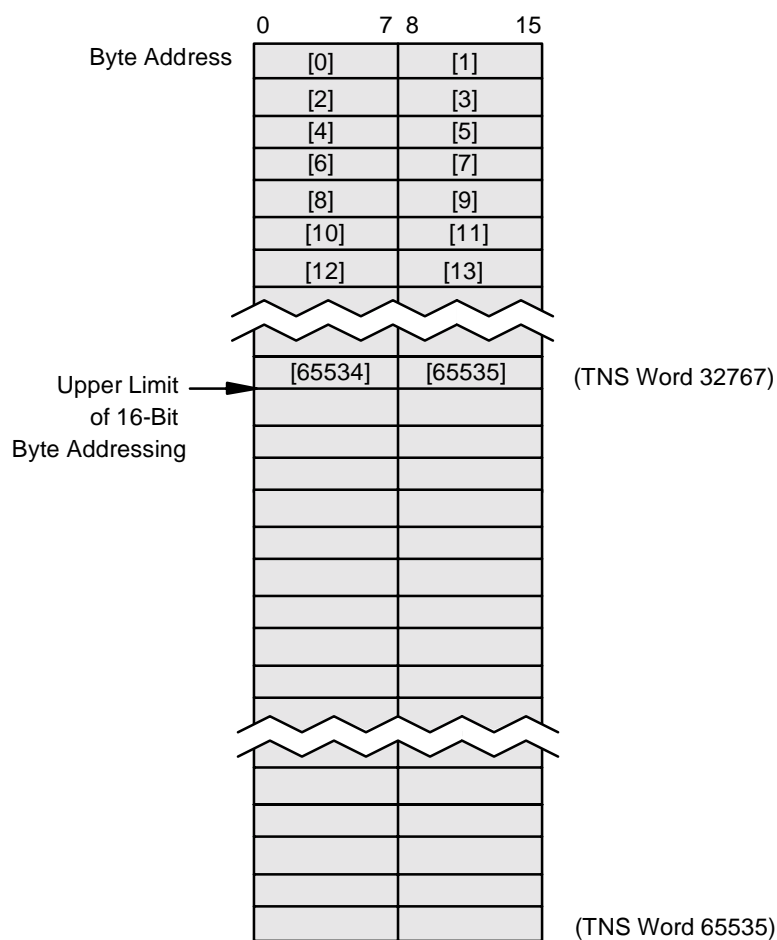
Bytes and multibyte strings can begin at any memory address; alignment on word boundaries is not required (as is required for all nonbyte data). Some locations cannot be reached by 16-bit addresses. In the user data segment, locations that are addressable by 16-bit addresses start at BYTE[0] and extend through BYTE[65535]. Two bytes are stored in each word; therefore the first 32,768 words of the data segment (WORD[0:32767]) can store 65,536 bytes. The upper half of the data segment, WORD[32768:65535], cannot be accessed with 16-bit addressing; this area is byte addressable only with the use of extended addressing.

In a code segment, byte addresses are computed relative to whether the current setting of the P (for program counter) register is in the lower or the upper half of the code segment. Therefore, the entire code segment (WORD [0:65535]) is byte addressable—though limited in range to half a segment at a time.

For 16-bit byte addressing of a data segment, shown in [Figure 3-3](#), bytes 0 and 1 are in the first TNS word of the segment, and 65,534 and 65,535 are in TNS word 32,767. In the case of a TNS code segment, bytes 0 and 1 could be either in TNS word 0 (if the

current P register value is less than 32,768) or in TNS word 32,768 (if the current P register value is 32,768 or higher).

**Figure 3-3. Byte-Manipulating Instructions Assume Big-Endian Numbering**



# Instructions for Unsigned and Signed Arithmetic

TNS instructions that perform logical and integer arithmetic operations are named according to TNS data lengths: TNS word, TNS doubleword, or TNS quadrupleword. Unsigned integer arithmetic is restricted mostly to TNS word operands. Signed integer arithmetic can be performed on all three formats.

The TNS instruction set provides arithmetic on both signed and unsigned numbers. **Signed numbers** are characterized by being able to represent both positive and negative values; **unsigned numbers** represent only positive values. Floating-point numbers, described in the next topic, are always signed.

The representable range of numbers is determined by the sizes of operands (that is, word, doubleword, and quadrupleword). The ranges are shown in [Table 3-1](#).

Whether a word operand is treated as a signed or an unsigned value is determined by the instruction used when a calculation is performed.

---

**Table 3-1. Ranges of Numbers Representable by Different Data Lengths**

Data Type	Range of Numbers Represented
Single TNS word	–32,768 through +32,767 0 through 65,535 (unsigned)
TNS doubleword	–2,147,483,648 through +2,147,483,647
TNS quadrupleword	–9,223,372,036,854,775,808 through +9,223,372,036,854,775,807

---

## Unsigned Arithmetic

Operations on unsigned numbers are mostly restricted to 16-bit quantities (exceptions are the doubleword shift instructions, DLLS and DLRS). Unsigned arithmetic is indicated by the execution of **logical instructions**. The logical instructions provide for arithmetic (LADD, LADI, LSUB, LMPY, LDIV), comparison and two's-complement negation (LCMP, LNEG), Boolean logic operations (LAND, LOR, XOR, DPF), and 16-bit or 32-bit shifts (LLS, LRS, DLLS, DLRS). The logical multiply instruction (LMPY) returns a doubleword product, and the logical divide instruction returns a two-word unsigned quotient with remainder.

The results obtained from a logical add or subtract (LADD or LSUB) are identical to those obtained from integer add or subtract (see next subheading) except that logical add and subtract never change the Overflow indicator and so these operations never trap. (The 16-bit result, the Condition Code setting, and the Carry indicator setting are the same.) Logical divide (LDIV), however, sets the Overflow indicator if the quotient cannot be represented in 16 bits (unsigned).



## Signed Arithmetic

Signed numbers are represented in 16 bits (a TNS word), 32 bits (TNS doubleword), or 64 bits (TNS quadrupleword).

Positive values are represented in true binary (base 2) notation. Negative values are represented in **two's-complement** notation with the sign bit of the most significant word set to 1 (that is, WORD[0].<0>). The two's complement of a number is obtained by inverting each bit position in the number, then adding a 1. For example, in 16 bits, the number 2 is represented:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

and the number -2 is represented:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
```

Signed arithmetic is indicated by the execution of **integer instructions**. The integer instructions for single words provide for integer arithmetic (IADD, ADDI, ADDM, ADAR, ADRA, ADXI, ISUB, SBAR, SBRA, IMPY, IDIV), comparison and two's complement negation (ICMP, CMPI, INEG), special load operations (LDI, LDLI, LDXI, LDRA, STAR), and arithmetic shifts (ALS, ARS).

Similarly, instructions are provided for integer arithmetic on doubleword operands (DADD, DSUB, DMPY, DDIV), comparison and negation (DCMP, DNEG), special load and test operations (MOND, ZERD, ONED, DTST), and arithmetic shifts (DALS, DARS).

Six instructions for quadrupleword operands provide for integer arithmetic (QADD, QSUB, QMPY, QDIV) and for comparison and negation (QCMP, QNEG). In addition, three instructions (QUP, QDWN, QRND) provide utility operations when used as part of a decimal arithmetic computation (see next topic).

For all integer add, subtract, and negate instructions, the result on an overflow is the truncated sum or difference. For multiplication and division instructions, the result on overflow is undefined and will differ from results produced by other TNS or TNS/R processors. When an overflow occurs, the **Overflow** (V) indicator is set and (if the overflow trap is currently enabled) an interrupt to the operating system overflow interrupt handler occurs. An overflow condition also occurs if a divide operation is attempted with a divisor of 0. When no overflow occurs in these signed instructions, V is reset to 0. (Any instruction that cannot set V always leaves V unchanged.)

In addition to the Overflow indicator, two other indicators are subject to change as the result of an arithmetic operation. They are:

- **Condition Code** (CC). This generally indicates whether the result of a computation or a load operation was a negative value, zero, or a positive value. In cases where overflow occurred, CC reflects the truncated result.
- **Carry** (K). This indicates that a carry out of the high-order bit position occurred on an add instruction. For subtract instructions, K=1 indicates that a borrow out of the high-order bit position did not occur; that is, A-B sets K only if A '>=' B. For multiply and divide instructions K is meaningless and is reset to 0.

All three indicators (V, CC, and K) can be tested by associated test-and-branch instructions (branch-on-overflow, branch-on-condition-code, and branch-on-carry) and the program execution sequence can be altered accordingly.

## Instructions for Decimal and Floating-Point Arithmetic

Decimal arithmetic can be performed only on quadrupleword operands. Two groups of floating-point instructions permit use of both doubleword and quadrupleword operands. Because the exponent field is the same size in both formats, the range of representable numbers is the same; the extended format, however, provides greater precision (in the fraction field).

### Decimal Arithmetic

Decimal (base 10) arithmetic is provided indirectly. Decimal operations are performed by explicitly converting operands from ASCII-digit string form to 64-bit binary form, doing binary integer arithmetic, and then converting the result back to ASCII-digit string form.

The quadrupleword operands used in the signed-arithmetic computation can represent 19-digit numbers in the range shown in the table in the preceding topic.

The following instructions are provided for converting operands between quadrupleword and integer, logical, and doubleword data formats: CQI, CQL, CQD, CIQ, CLQ, CDQ.

In addition, there are eight instructions (CFQ, CFQR, CEQ, CEQR, CQF, CQFR, CQE, CQER) that provide conversions between quadrupleword operands and both forms of floating-point operands (see [Floating-Point Arithmetic](#), next), either with rounding or without rounding.

### Floating-Point Arithmetic

NonStop S-series servers support two schemes for floating-point arithmetic, IEEE and TNS, using separate ways of encoding radix-2 floating-point numbers into 32-bit and 64-bit values, and using separate operations and separate algorithms. In TNS and accelerated TNS programs, the only supported forms of floating-point arithmetic are the legacy TNS floating-point formats, as used in the original NonStop systems. The basic operations of floating add, multiply, compare, and so on, are implemented by millicode routines, not by processor hardware. Native mode programs have a compile-time choice between using the legacy TNS formats or using the industry standard IEEE formats (IEEE Standard 754-1985). The IEEE scheme has much higher performance and better numerical properties; the TNS scheme is data compatible with earlier NonStop systems. IEEE-format values use the RISC processor's floating-point registers and floating-point circuits. Native use of TNS-format values is implemented by integer operations in millicode routines, as in TNS and accelerated modes, with no use of the RISC processor's floating-point registers or circuits.

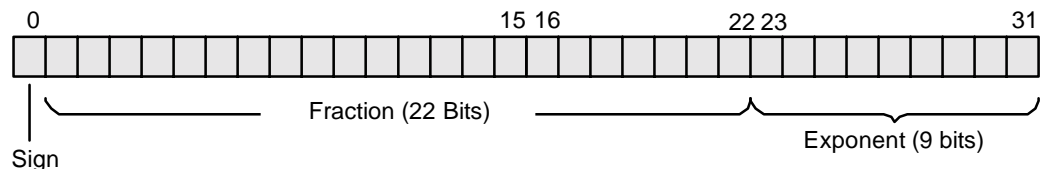
The bit-level representations of TNS and IEEE floating-point values are incompatible. The two formats have different precisions and exponent ranges in their 32-bit sizes, and different precisions and exponent ranges in their 64-bit sizes.

For a description of the IEEE representations as used on NonStop S-series servers, see Chapter 7, “FPU Overview,” of *MIPS RISC Architecture* by Gerry Kane and Joe Heinrich.

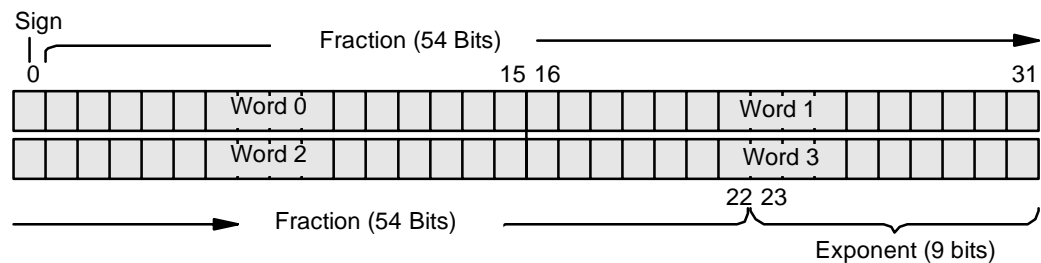
The fraction of a TNS-format floating-point number is always normalized, to be greater than or equal to 1 and less than 2. The high-order integer bit is therefore dropped and assumed to have the value of 1. During all calculations, the sign is temporarily removed and the assumed integer bit reinserted. The integer plus the 22 fraction bits of a floating-point number are equivalent to 6.9 decimal digits; the 55 bits of an extended floating-point number are equivalent to 16.5 decimal digits. (Both formats are shown in [Figure 3-4](#).) If the value of the number to be represented is zero, the sign is 0, the fraction is 0, and the exponent is 0.

**Figure 3-4. Formats for TNS Floating-Point and Extended Floating-Point Data**

TNS-format floating-point doubleword:



TNS-format extended floating-point quadrupleword:



VST230.vsd

The fraction of a floating-point number is a binary number with the binary point always between the assumed integer bit and the high-order fraction bit. The exponent part of the number, bits 7 through 15 of the low-order word, indicates the power of 2 multiplied by 1 plus the fraction. This field can contain values from 0 through 511. To express numbers of both large and small absolute magnitude, the exponent is expressed as an excess -256 value; that is, 256 is added to the actual exponent of the number before it is stored. The exponent range is therefore actually -256 through +255.

The sign of a floating-point number is explicitly given in the high-order bit; a 0 is positive, and a 1 is negative. Unlike integers, which use two's complement, negating a

floating-point number affects only the high-order bit and not the fraction or exponent fields.

The absolute-value range of 32-bit floating-point numbers is  $\pm 2^{-256}$  through  $\pm (1 - 2^{-23}) * 2^{256}$ . In decimal notation, this is approximately  $\pm 8.64 * 10^{-78}$  through  $\pm 1.15 * 10^{77}$ .

For extended floating-point numbers (64 bits), the range is the same; only the precision is increased:  $\pm 2^{-256}$  through  $\pm (1 - 2^{-55}) * 2^{256}$ . (Note, however, that the value  $+2^{-256}$  is not representable; it would look like 0 in either floating-point or extended floating-point format.)

For floating-point and extended floating-point arithmetic, the Overflow (V) indicator is set if the exponent becomes either greater than +255 (exponent overflow) or less than -256 (exponent underflow) during an attempt to represent the normalized result of some operation. Otherwise, V is reset to 0. If the divisor in a divide operation is 0, the Overflow indicator is also set. If any conversion instruction causes a numeric overflow ("illegal conversion"), the Overflow indicator is set and the result (including Condition Code) is undefined. If the result of some operation has a zero fraction and nonzero exponent or sign, the value is forced to zero.

[Table 3-2](#) defines termination conditions for various floating-point arithmetic errors. (For further explanation of the Condition Code, refer to the topic [The Environment Register](#) on page 6-6.)

---

**Table 3-2. Termination Codes for Floating-Point Arithmetic Errors**

Condition	Overflow	CC	Result
Exponent overflow	1	00	Calculated result with exponent truncated
Exponent underflow	1	10	Calculated result with exponent truncated
Divide by zero	1	01	Zero
Illegal conversion	1	xx	Undefined

---

# Memory Addressing and Access

This section describes how memory is addressed and accessed within the processors of the NonStop S-series servers.

---

**Note.** Some very low-level detail is presented in this section. Be aware that specific address allocations, table formats, and many architectural details have changed in the past and will change in the future. Application software should never directly use or depend on the details presented in this section. Applications that require memory management must always use the appropriate callable procedures. Such procedures accommodate the differences in low-level operations.

---

The first fifteen topics in this section cover the addressing principles. The remaining topics cover the access logic.

- [The Process Address Space](#)
- [Organization of the Process Address Space](#)
- [Addressing in the Process Address Space](#)
- [Address Formats](#)
- [Selectable and Flat Logical Segments](#)
- [First Four Relative Segments](#)
- [Main Stack and SRL Data](#)
- [Last Eight Regions](#)
- [Native Process Code Allocations](#)
- [TNS Process Code Allocations](#)
- [Allocation for TNS and Accelerated Code](#)
- [Chart of Nonprivileged Space Allocation](#)
- [Physical Memory Addressing](#)
- [Kseg0 Usage](#)
- [Kseg2 Usage](#)
- [Kseg1 Memory Access](#)
- [Kseg0 Memory Access](#)
- [Kseg2 and Nonprivileged Space Memory Access](#)
- [The TLBPID Process Identifier](#)
- [Nonglobal Address Translation](#)
- [Address Translation of Global Elements](#)
- [Address-Mapping Tables](#)
- [Access of Special Pages](#)
- [Defining Unallocated Space](#)
- [Context-Bound Addresses](#)

# The Process Address Space

The virtual memory system of each processor in the NonStop S-series servers provides a 4-GB virtual address space, as illustrated in [Figure 4-1](#). Virtual memory is arranged as multiple **process address spaces**. These address spaces are primarily assigned to the various processes executing in the processor. However, only one such space can be the current one.

These spaces are not exclusively for “processes”—the operating system claims one process address space for use by interrupt handlers (which are not processes).

The 256 most recently active process address spaces (including the current one) have special identifiers that make them eligible to have their relative virtual addresses translated to physical addresses.

Each process address space is divided into a **nonprivileged space** (the lower 2 GB of virtual addresses) and a **privileged space** (the higher 2 GB of virtual addresses). Each of these spaces can have areas that are either global or nonglobal. **Global addresses** have the same data in all process address spaces.

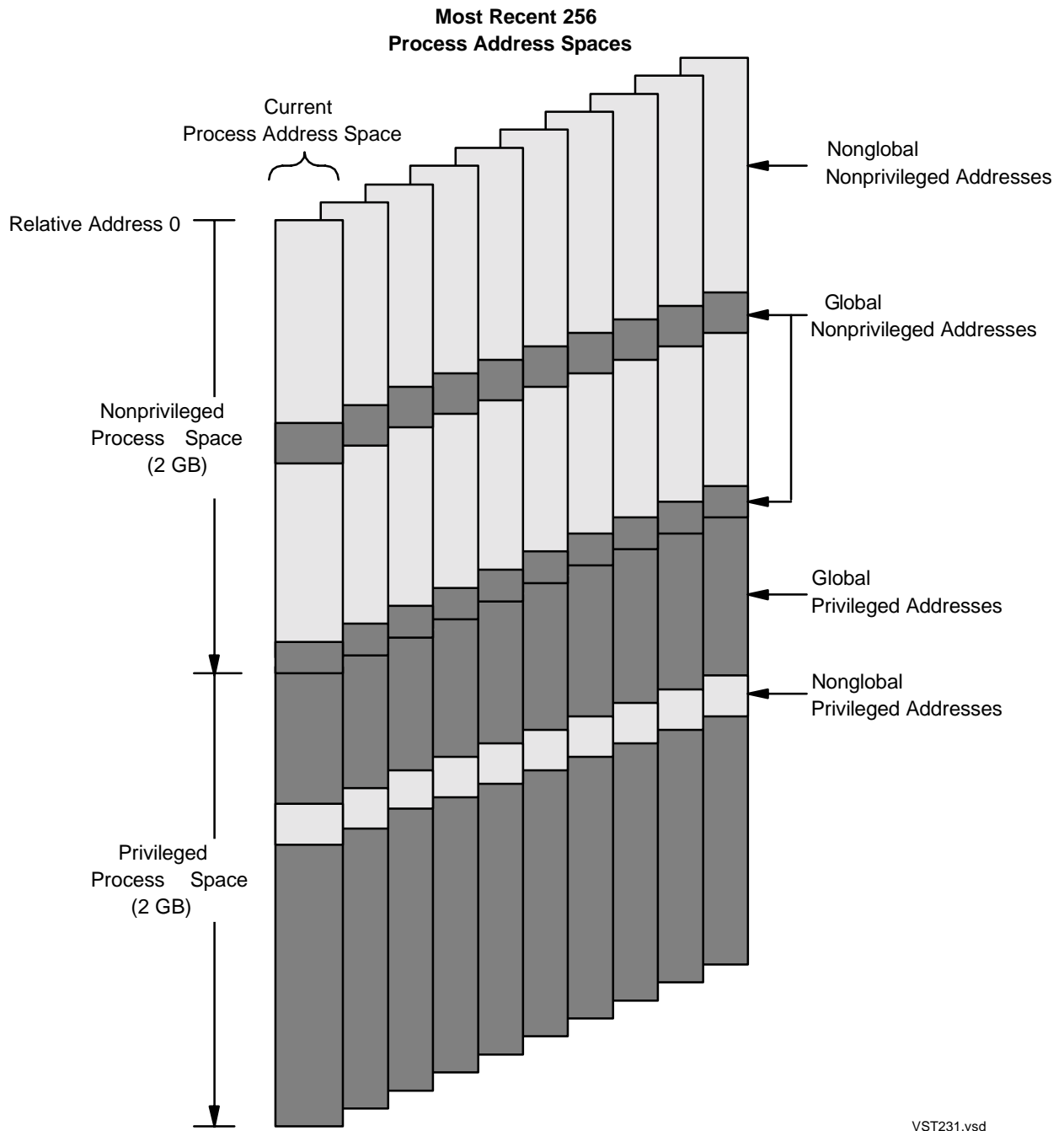
Privileged addresses can be accessed only by processes that are executing in privileged mode. The privileged space, being mostly a global space, is used to address system elements. Such elements include physical memory, resident privileged procedures, and system data. Much of the privileged address space is devoted to providing “absolute” addresses for segments that also have addresses in nonprivileged space.

The nonprivileged space is used primarily to address the code and data for a given process. Such code and data might or might not be shared among processes, depending on specific applications. However, those elements that are designated as *global* are always shared. These would include all nonprivileged or nonresident system library procedures

---

**Note.** In publications describing RVUs prior to G05.00, the process address space was referred to as the process data space, and nonprivileged space and privileged space were referred to as user space and kernel space, respectively. These terminology changes reflect current industry standard usage.

---

**Figure 4-1. Nonprivileged and Privileged Spaces Include Global and Nonglobal Areas**

# Organization of the Process Address Space

As [Figure 4-2](#) indicates, all of the nonprivileged space and half of the privileged space are subdivided for addressing purposes into three substructures. (The first half of the privileged space, shaded dark, does not use this same organization, as explained further in the next two topics.)

For those areas of the process address space that use the organization shown in [Figure 4-2](#), the major organizational division is the **region**. There are 64 regions in the nonprivileged space and 32 regions in the higher-address half of the privileged space. The entity called a region exists primarily to reduce the amount of memory consumed by the addressing tables. That is, segment tables need not exist for unallocated regions.

Although there are 64 regions available in the nonprivileged space, processes typically use less than half of them.

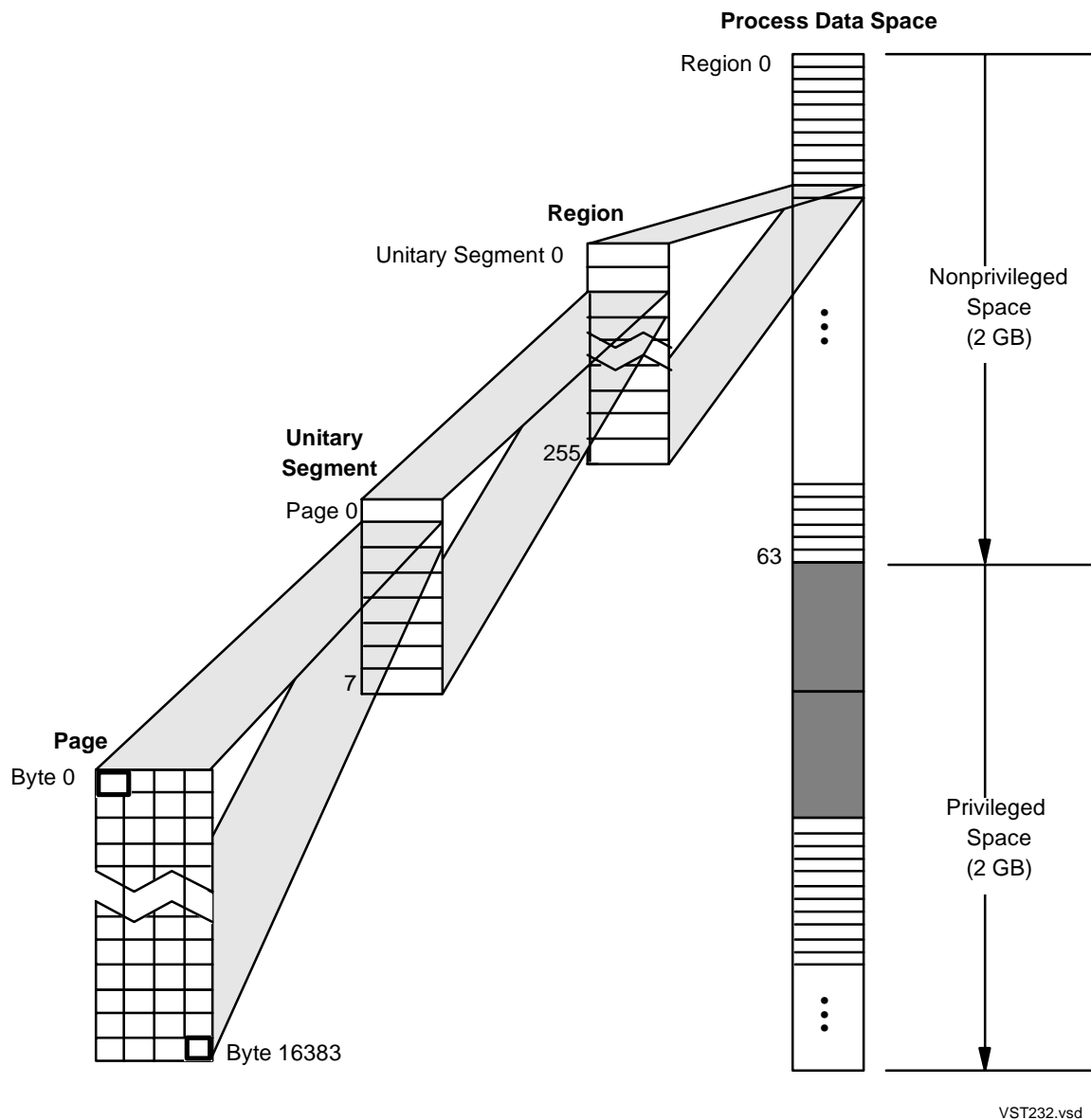
Although regions can be numbered decimally (such as 0 through 63, shown in the figure for the nonprivileged space), the more useful reference is the hexadecimal starting address. Thus, for example, “the 5E region” is the one that starts at hexadecimal address 5E000000. Following this convention, “the 50 region” is spoken of as the “five-zero region,” rather than “region fifty,” to avoid the implication of decimal numbering.

A region is 32 megabytes, or 256 **unitary segments**. Thus in the entire nonprivileged space there are 16,384 unitary segments (64 times 256). The adjective “unitary” is used to distinguish this special use of the term “segment” from the primary use, specifically “logical segment,” an application programming entity. The logical (or “extended”) segment consists of one or more unitary segments. The unitary segment is mostly a memory-management entity. The term is sometimes abbreviated to “unitseg.”

A unitary segment is 128 kilobytes, or eight **pages**.

A page is 16,384 bytes of information. When stored in physical memory, a page occupies one 16,384-byte physical **frame** of storage space.



**Figure 4-2. Process Address Space Includes Regions, Unitary Segments, and Pages**

# Addressing in the Process Address Space

All addressing is accomplished with 32-bit byte addresses, thus determining the 4-gigabyte addressing range of the process address space. That space, however, is not of uniform function. As already shown by the previous two topics, the process address space is divided into a nonprivileged space (the lower-address half) and a privileged space (the higher-address half). The privileged space can be addressed only in privileged mode; the nonprivileged space can be addressed in either privileged or nonprivileged mode.

A further division of the process address space divides the privileged space into three distinct areas, as shown in [Figure 4-3](#). These are designated Kseg0, Kseg1, and Kseg2. Kseg0 and Kseg1 together constitute the lower-address half of the privileged space, and Kseg2 constitutes the higher-address half. Therefore, there are altogether four addressing spaces in the process address space. (Not used is a RISC chip capability that permits splitting the Kseg2 area into Kseg2 and Kseg3 areas.) The Kseg nomenclature is RISC terminology and has nothing to do with “segments” in NonStop processors.

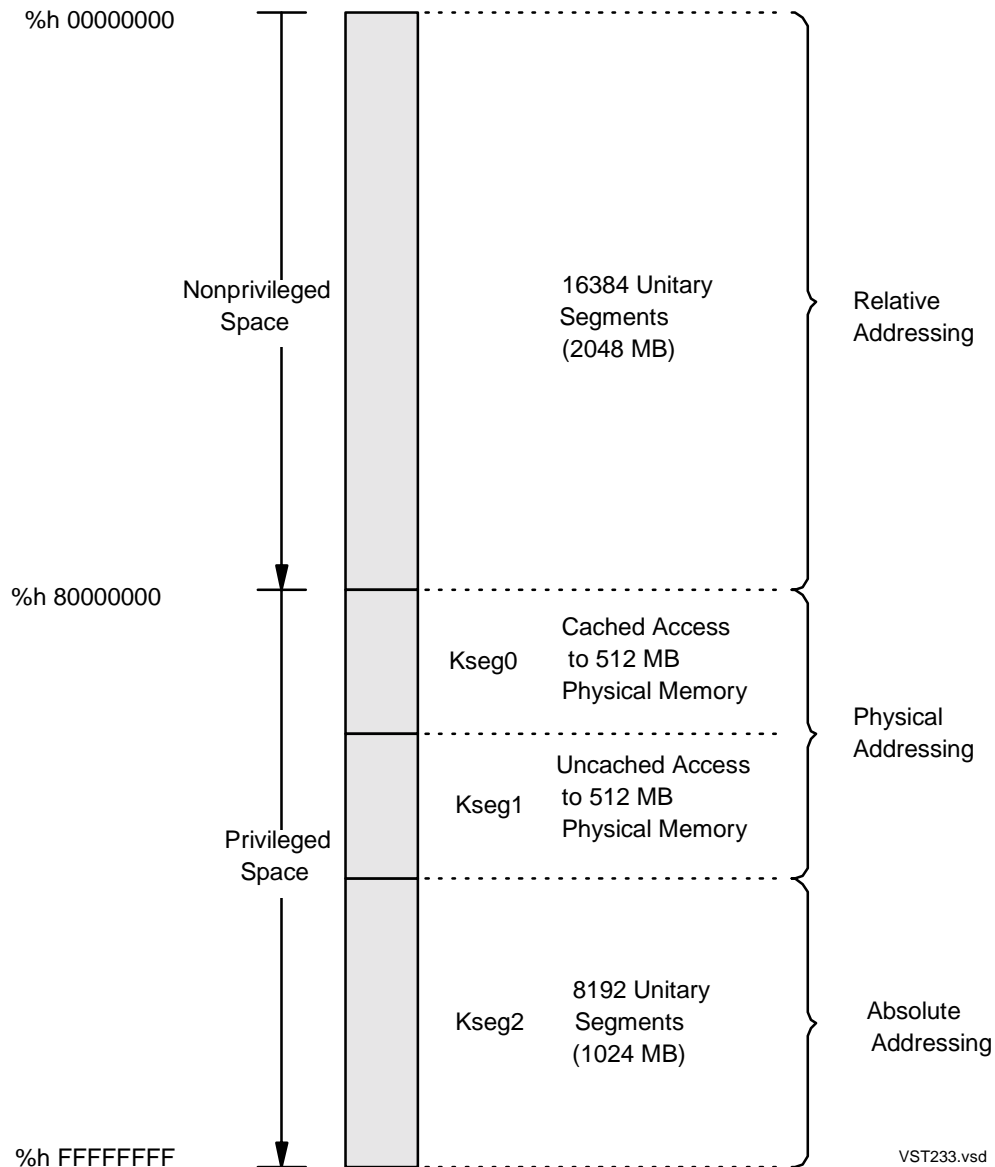
The first of the four spaces, consisting of the entire nonprivileged space, is addressed with **relative addressing**. That is, every process can address its own code and data in pages and unitary segments relative to page 0 in segment 0. Accordingly, the term “process-relative addressing” is sometimes used for added clarity.

The second of the four spaces, Kseg0, corresponds directly to the physical memory addresses, up to a maximum of 512 megabytes. This space is addressed with **physical addressing**. The availability of physical addressing in Kseg0 provides a means for privileged software to directly access physical memory without the need for low-level translation of addresses.

Kseg1 is very similar to Kseg0, except that it addresses the same physical memory even more directly. That is, it bypasses the caches. This mode of addressing is used only in special circumstances, such as reading and writing special hardware registers that are addressed as if they are memory cells (**memory-mapped registers**).

Kseg2 provides privileged addressing that can be mapped to different areas of physical memory as required. One region (256 unitary segments) is used for privileged data for the process. The rest of Kseg2 provides **absolute addressing** for up to 7936 (that is, 8192 less 256) absolute unitary segments.

All Kseg0 and Kseg1 addresses map directly to physical addresses, so they are always global. Each page of nonprivileged space and Kseg2 addresses can be mapped to any frame of physical memory, and each mapping can be either per-process or global. All Kseg2 addresses are global.

**Figure 4-3. Four Distinct Addressing Areas Exist in the Process Address Space**

# Address Formats

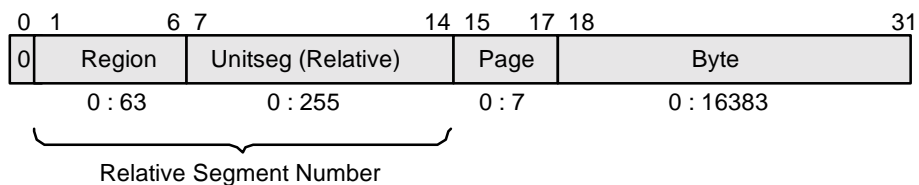
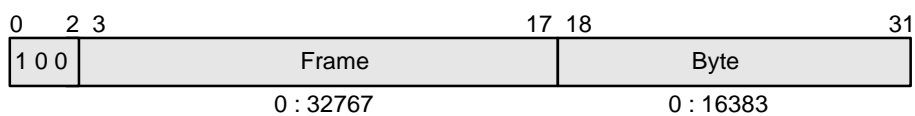
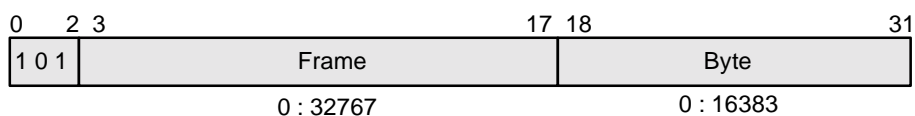
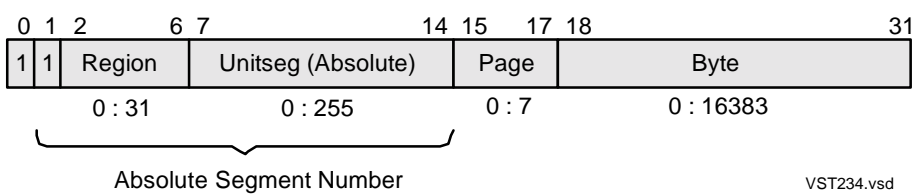
The 32-bit address used for each of the four addressing spaces is identified by the bit pattern of the first one, two, or three bits. See [Figure 4-4](#). The one-bit pattern of “0” specifies a nonprivileged space address (relative). The three-bit pattern of “100” or “101” specifies a Kseg0 or Kseg1 address, respectively (physical). The two-bit pattern of “11” specifies a Kseg2 address.

Any address with a 0 in the most significant bit position is a nonprivileged space address. All the remaining bits are then available to specify a particular byte within that space. Bits 1 through 6 specify a region, 7 through 14 specify a unitary segment, 15 through 17 a page, and 18 through 31 a byte. Any address with a 1 in the most significant bit position is in the higher-address half of the process address space and therefore in the privileged space.

In nonprivileged space, it is sometimes convenient to ignore the region boundaries and treat bits 1 through 14 of the relative address as a **relative segment number**.

All addresses in the privileged space begin with a 1 in the most significant bit position. If the next-most significant bit is a 0, the address is a physical address, either within the Kseg0 space or within the Kseg1 space. If the third-most significant bit is a 0, the address is within Kseg0; if that bit is a 1, the address is within Kseg1. In either case, the field consisting of bits 3 through 17 specifies a frame in physical memory, and the field consisting of bits 18 through 31 specifies the byte within that frame. (Physical memory does not use the concept of segments, as virtual memory does.)

If the two most significant bits are both 1, the address is within the Kseg2 space. Like the nonprivileged space, Kseg2 addresses are organized into regions, unitary segments, and pages. Bits 2 through 6 specify a region, 7 through 14 specify a unitary segment, 15 through 17 a page, and 18 through 31 a byte. (Unlike the nonprivileged space, only five bits are available to specify a region, because bit 1 is used to distinguish the Kseg2 address from Kseg0 and Kseg1 addresses; thus 32 regions are available instead of 64.)

**Figure 4-4. Address Formats Are Slightly Different for Each Addressing Area****Nonprivileged Space Address****Kseg0 Address****Kseg1 Address****Kseg2 Address**

VST234.vsd

# Selectable and Flat Logical Segments

Data address space for a process is allocated in virtual memory by one or more `SEGMENT_ALLOCATE_` procedure calls. Each call allocates a **logical segment** that is of a specifiable size.

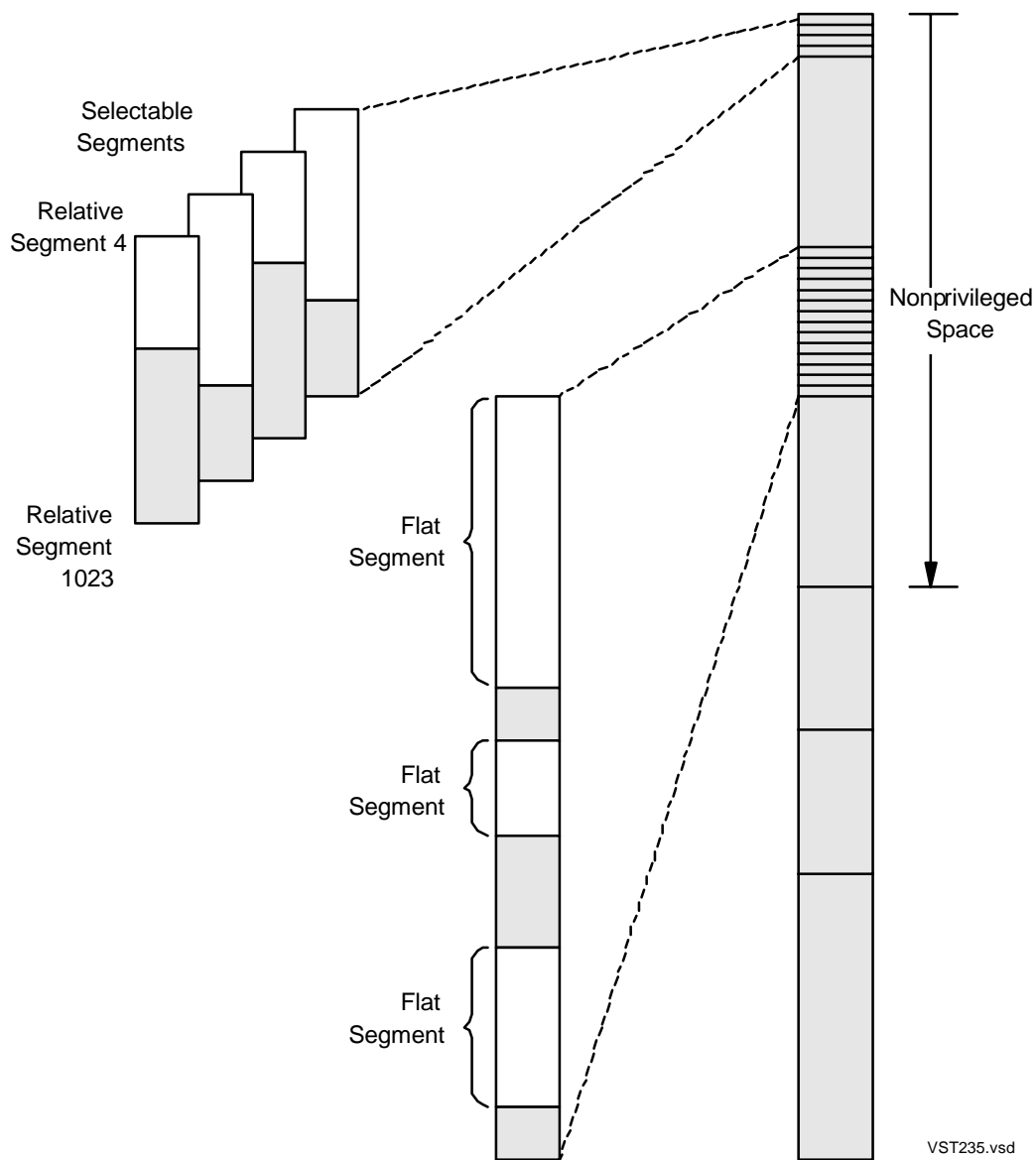
A logical segment can be either one of two kinds: a **selectable segment** or a **flat segment**. These are compared in [Figure 4-5](#), and the choice is made by a parameter supplied to the `SEGMENT_ALLOCATE_` call. (A selectable segment previously was sometimes called an extended data segment or extended segment.)

Any number of selectable segments can be allocated; however, they all share the same address space, hexadecimal addresses 00080000 through 07FFFFFFF. Thus, only one selectable segment can be in use at any given time. The selection is made with a `SEGMENT_USE_` procedure call.

The maximum size of a selectable segment is 127.5 megabytes, the first half megabyte being reserved for compatibility with earlier TNS systems (see next topic). All selectable segments begin at the start of relative unitary segment 4 and can include, as a maximum, relative unitary segment 1023, as shown in the figure.

Flat segments, on the other hand, have distinct addresses. The operating system allocates distinct addresses within the nonprivileged area starting at a fixed upper boundary of 4E000000 and proceeding downward to a variable lower boundary. Flat segments can theoretically be up to 1120 megabytes in size and need not begin on a region boundary.

Because flat segments have distinct addresses, they do not need to be made current with a `SEGMENT_USE_` procedure call. They are continuously available to the process.

**Figure 4-5. Selectable and Flat Logical Segments Differ in Allocation Method**

# First Four Relative Segments

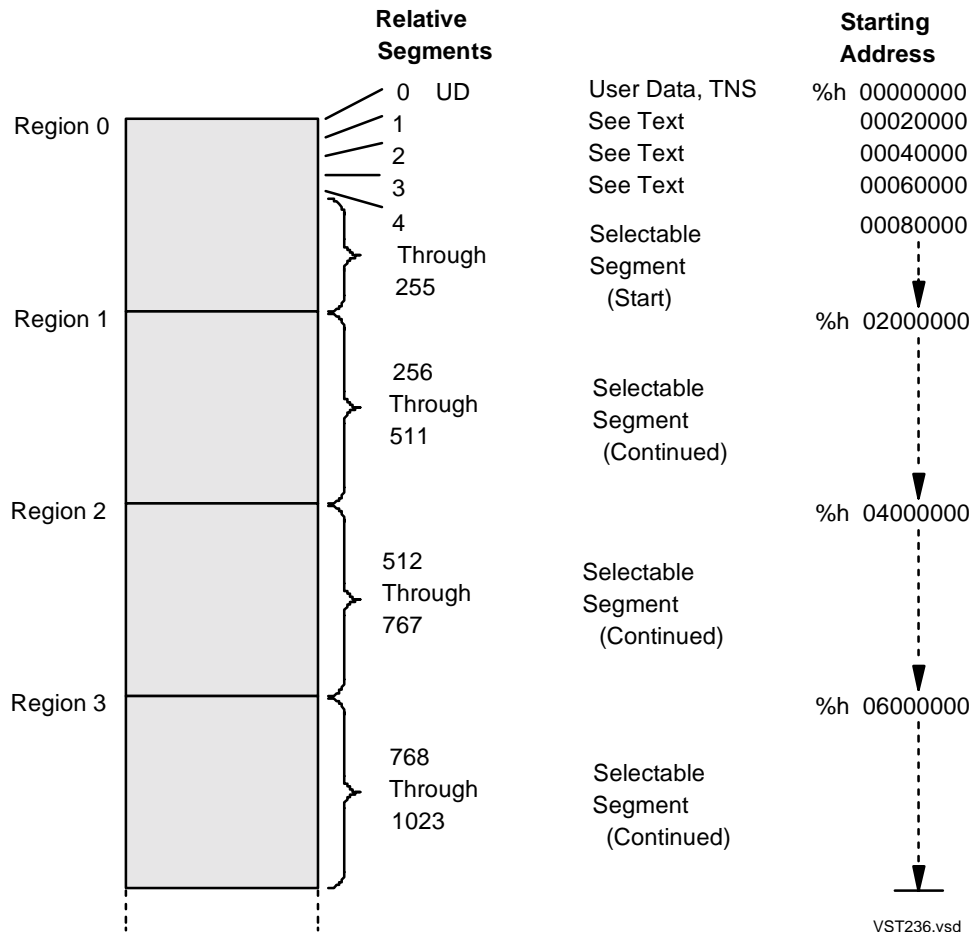
[Figure 4-6](#) shows addressing details for the first four regions, particularly the first region. Hexadecimal starting addresses are indicated for the regions and the first five relative segments, 0 through 4.

Relative segment 0 is the TNS user data segment, which, for TNS processes, provides relative addressing for process global variables and the TNS data stack. Relative segments 1, 2, and 3 are skipped. Then, the current selectable segment begins at relative segment 4 (hexadecimal address 00080000).

The reason for skipping relative segments 1, 2, and 3 is that these segment numbers are reserved for backward compatibility. In the TNS architecture, these are assigned to **system data segment**, **current code segment**, and **latest user code segment**, respectively. In the NonStop S-series processors, most references to system data are through Kseg0 addresses, and thus references to relative segment 1 are not usually needed. Similarly, current code and user code (that is, the last-used user code segment) also are not usually needed, because all code segments are simultaneously available in the NonStop S-series processors. Thus these three relative segment numbers are not normally used, but they do work—for those TNS applications that have not been converted to remove hard-coded references.

Because there are 256 unitary segments in a region, a selectable segment that consists of more than 252 unitary segments crosses at least one region boundary.



**Figure 4-6. For TNS Compatibility, the First Four Relative Segments Are Special**

## Main Stack and SRL Data

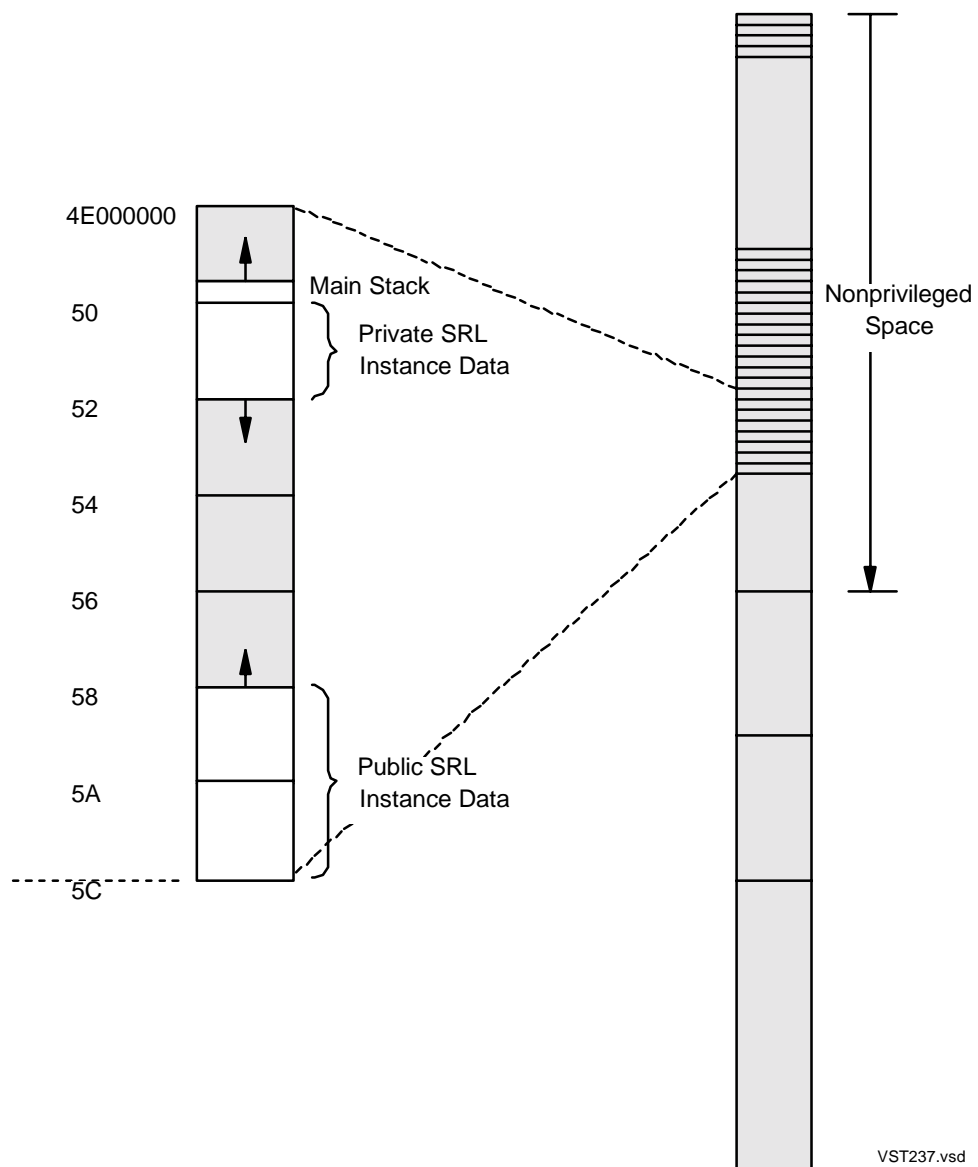
Following the space allocations for the user-allocatable flat segments are several regions that provide relative addressing for the main stack of the process and instance data for shared run-time libraries (SRLs). See [Figure 4-7](#). The 4E region (hexadecimal address 4E000000) is the beginning of this area, and the 5A region is the end of the area (address of the last byte is 5BFFFFFFF).

The main stack is allocated for all processes except privileged native processes. (Native mode and processes, as well as TNS modes and processes, are described in [Section 5, Instruction Processing Environments](#)). The reason for this exception is that privileged native processes start in the privileged state and continue to execute in that state for the rest of their existence. Thus they never need to use the main stack.

The main stack begins at the end of the 4E region and grows toward lower addresses. The main stack is said to originate at relative address 50000000. (The first byte within the stack area is thus 4FFFFFFF.)

If a process has private SRL procedures, instance data for these procedures is allocated upward from the start of the 50 region. Instance data for public SRL procedures, if any, is allocated downward from the end of the 5A region.

Other data for a process, such as the process file segment, privileged stack, and debug stack, are located in the third region of Kseg2, discussed later in this section.

**Figure 4-7. Main Stack and SRL Data for a Process Are in Nonprivileged Space**

# Last Eight Regions

All of the **user code**, and **user library**, and **shared run-time library (SRL)** allocations are accessed within the code regions, as illustrated in [Figure 4-8](#). These are the last eight regions (256 megabytes) in the nonprivileged space. In addition, the **system library** is also included in the code regions of each process.

With few exceptions (Debug, for example), writing is not permitted into any of the code regions. This restriction is under hardware control, cooperatively with system software. (However, privileged software can write anywhere that is physically addressable—through Kseg0 or Kseg1.)

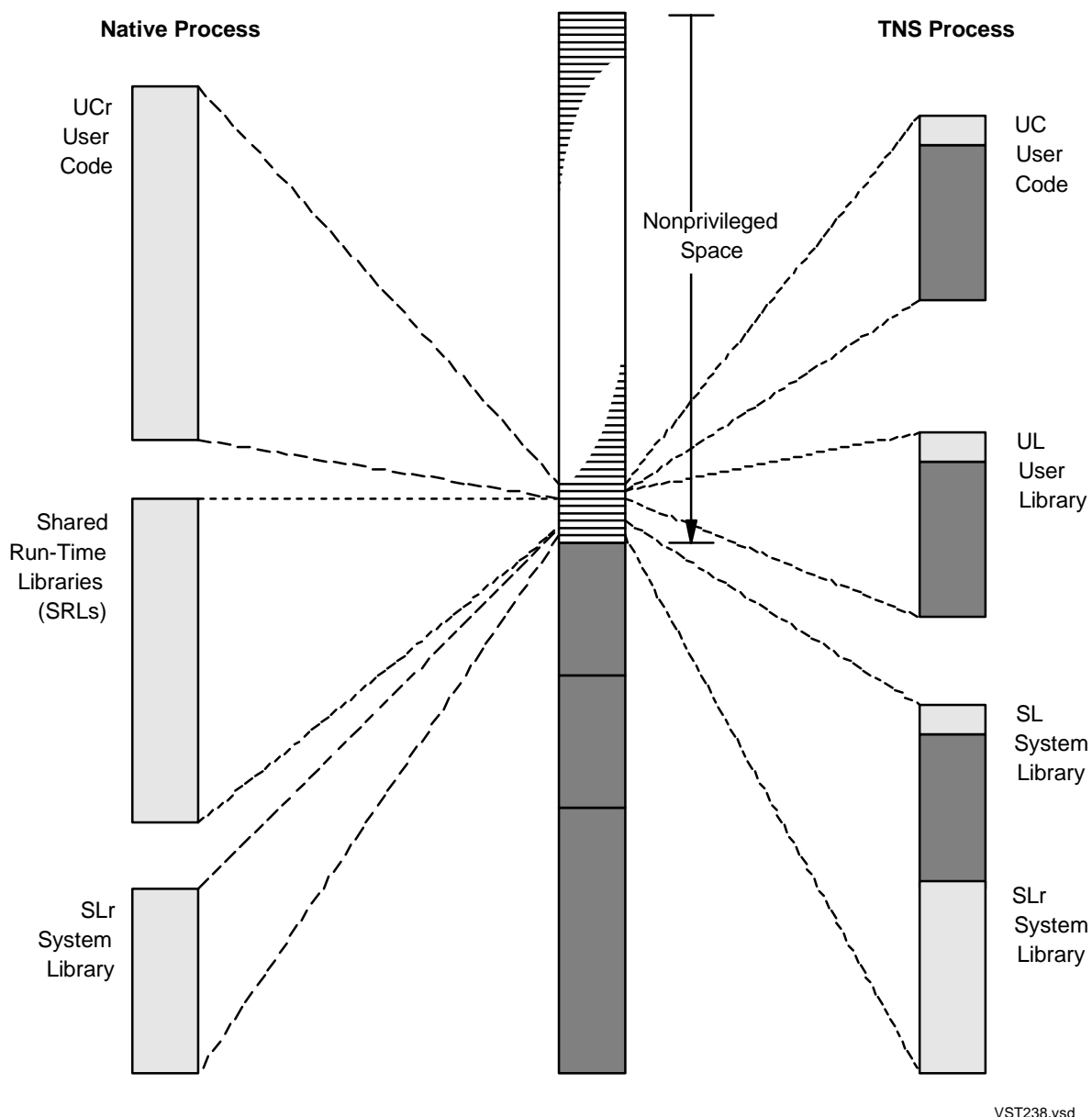
The number of regions allocated to the various kinds of code depends on the kind of process that was created: either native or TNS.

For a native process, two full regions (32 megabytes each) are available for user code allocation. User code allocation for a native process is designated as UC<sub>r</sub>, signifying “user code RISC.”

For a TNS process, one full region is dedicated to each kind of code area. That is, one region is for user code (UC), one for user library (UL). Although all regions have 256 segments, TNS architecture limitations restrict the maximum number of unitary code segments that can exist in each TNS code area. (Note lighter-shaded areas in the UC, UL, and SL areas.) The user code region can contain up to 32 unitary segments, and the user library can contain up to 32 unitary segments. The remainder of each of these regions (shaded darker in the figure) is available for accelerated code, if present.

The system library has one region available for native code, designated SL<sub>r</sub>, and one for TNS code, designated SL. Unlike system code, the system library (SL<sub>r</sub> and SL) contains some nonprivileged code and therefore must be included in the nonprivileged space. Much of the system library is actually coded as native procedures in the SL<sub>r</sub> region; calls to callable procedures from TNS or accelerated mode are routed through SL to SL<sub>r</sub> by to-RISC shells (discussed later).

System code (SC and SC<sub>r</sub>) is located in physical memory—specifically, in Kseg0 of the privileged space. The use of Kseg0 is described later, and so system code does not appear in this figure.

**Figure 4-8. The Last Eight Regions Are for Code Addressing**

# Native Process Code Allocations

For processes that have been compiled with a native mode compiler (a native process), the addressing for the user code file of that process starts at the beginning of the 70 region and can occupy all of the 70 and 72 regions.

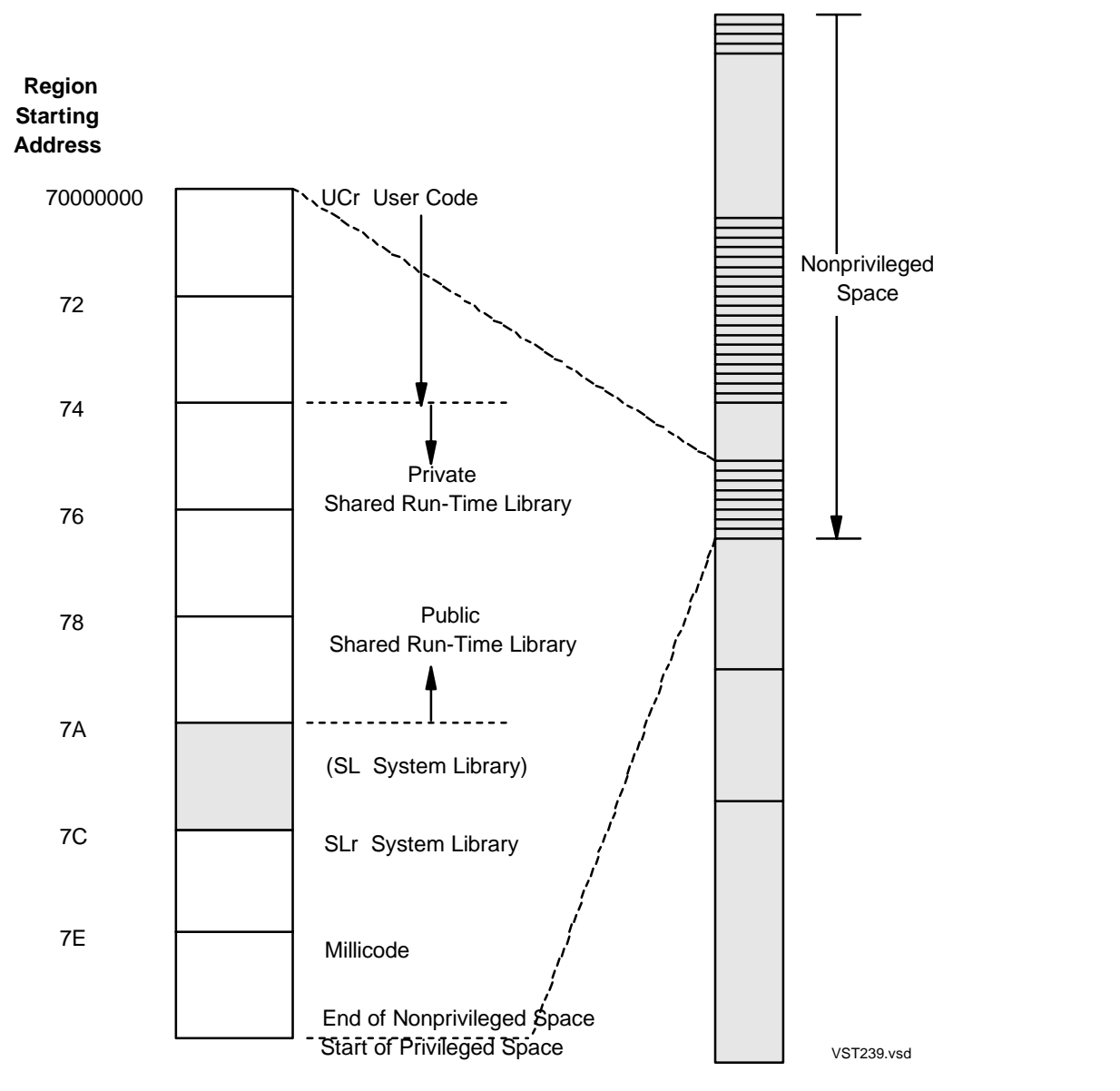
A native process may have procedures in a private shared run-time library (SRL). If so, the code for these is allocated from the start of the 74 region. The private SRL may be thought of as “native UL” or “ULr” (that is, native user library). In addition, the native process may also use public SRLs. The code for these is allocated downward from the end of the 78 region.

Addressing for the native object code of the system library for all processes occupies the 7C region. The system procedures that are located in this region are those that have one or both of the following characteristics: nonprivileged, nonresident. The system library and public SRL regions are globally mapped.

The 7E region, the last region of the nonprivileged space, contains parts of the millicode needed to support the currently executing process. That includes all of the nonprivileged portions of the millicode library executed by nonprivileged programs, including the TNS interpreter millicode. (Privileged parts of the millicode are located in the privileged space, Kseg0.) The last 64 unitary segments of the 7E region are used by the operating system for special purposes. Like the system library, the 7E region is globally mapped.

As shown in [Figure 4-9](#), the native mode user code allocations start at hexadecimal address 70000000. The system library allocations start at 7C000000. Private SRL code originates at 74000000 (upward) and public SRL code originates at 7A000000 (downward). Millicode allocations start at 7E000000. The SL system library is present in the 7A region, but it is not used by native processes.

Figure 4-9. Native Process Has Two Regions for User Code, Four for Libraries



# TNS Process Code Allocations

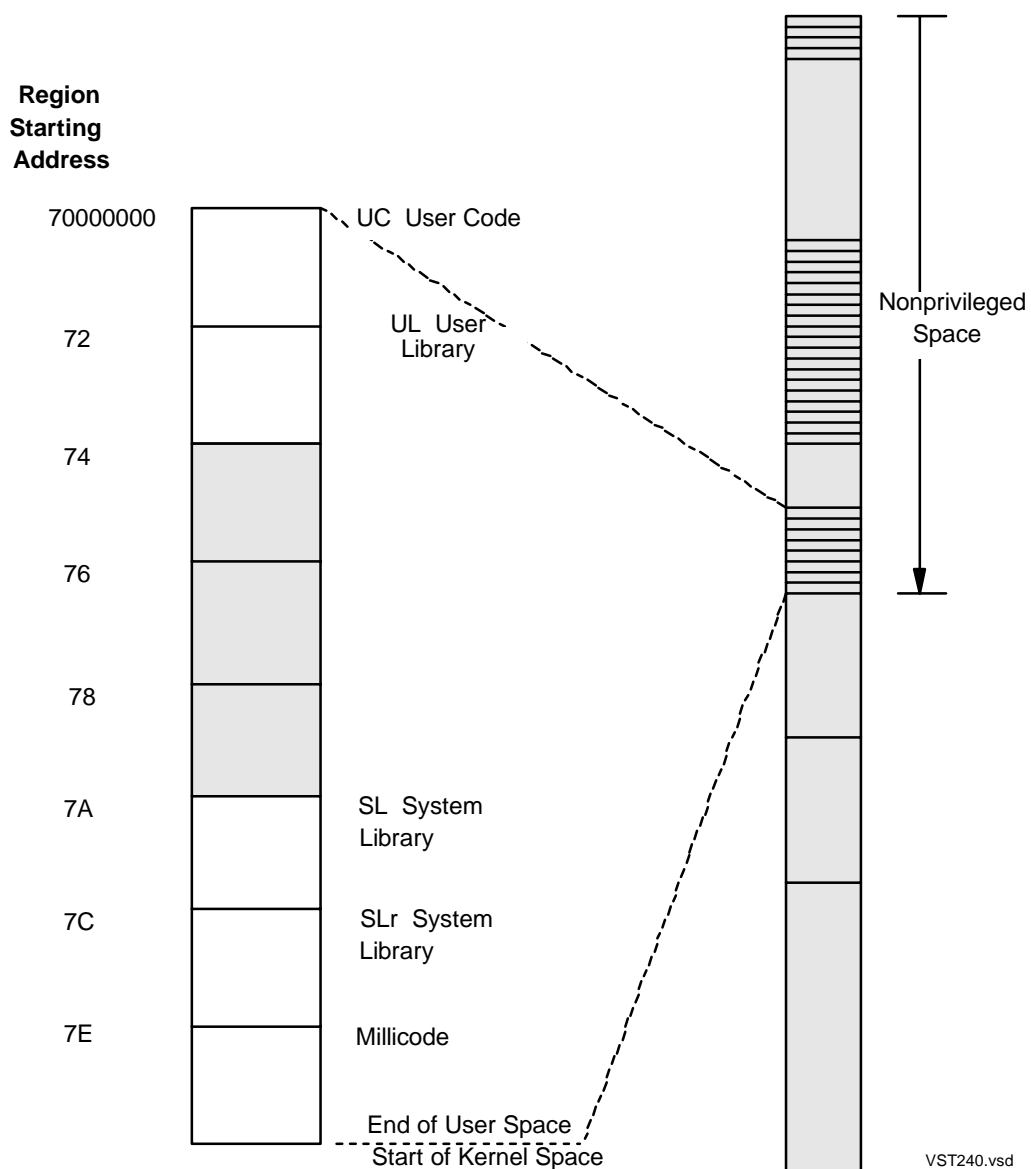
For TNS processes, including processes that have accelerated object code, addressing for the user code file of that process occupies the 70 region. The start of this region is hexadecimal address 70000000. See [Figure 4-10](#). Up to the first 32 (of the 256) unitary segments of this region can be allocated for the TNS object code. Corresponding accelerated code, if any, occupies a portion of the remaining 224 unitary segments; see next topic. The user code segments are usually designated as UC.0 through UC.31.

Addressing for the user library (if any) of a given process occupies the 72 region. The start of this region is hexadecimal address 72000000. Up to the first 32 unitary segments of this region can be allocated for the TNS object code. Corresponding accelerated code, if any, occupies a portion of the remaining 224 unitary segments; see next topic. The user library segments are usually designated as UL.0 through UL.31.

Addressing for the system library, which is common to all processes, occupies the 7A and 7C regions. Much of the actual code for the system library is in native code (SLr) and occupies the 7C region. However, TNS processes call the native library procedures through shells in the TNS area of the system library (SL). The start of this SL region is hexadecimal address 7A000000. Up to the first 32 unitary segments of this region can be allocated for the TNS object code. Corresponding accelerated code occupies a portion of the remaining 224 unitary segments; see next topic. The system library segments are usually designated as SL.0 through SL.31. The system library regions are globally mapped.

The 7E region, the last region of the nonprivileged space contains parts of the millicode needed to support the currently executing process. That includes all of the nonprivileged portions of the millicode library executed by nonprivileged programs, including the TNS interpreter millicode. (Privileged parts of the millicode are located in the privileged space, Kseg0.) The last 64 unitary segments of the 7E region are used by the operating system for special purposes. Like the system library, the 7E region is globally mapped.



**Figure 4-10. TNS Process Has One Region for User Code, One for User Library**

## Allocation for TNS and Accelerated Code

[Figure 4-11](#) shows an expansion of a code region for a process that uses accelerated object code. (This does not apply to native processes.) This could be a user code, user library, or system library region of any given nonprivileged space. The unitary segment numbers and starting addresses shown within the region are relative to the beginning of the region.

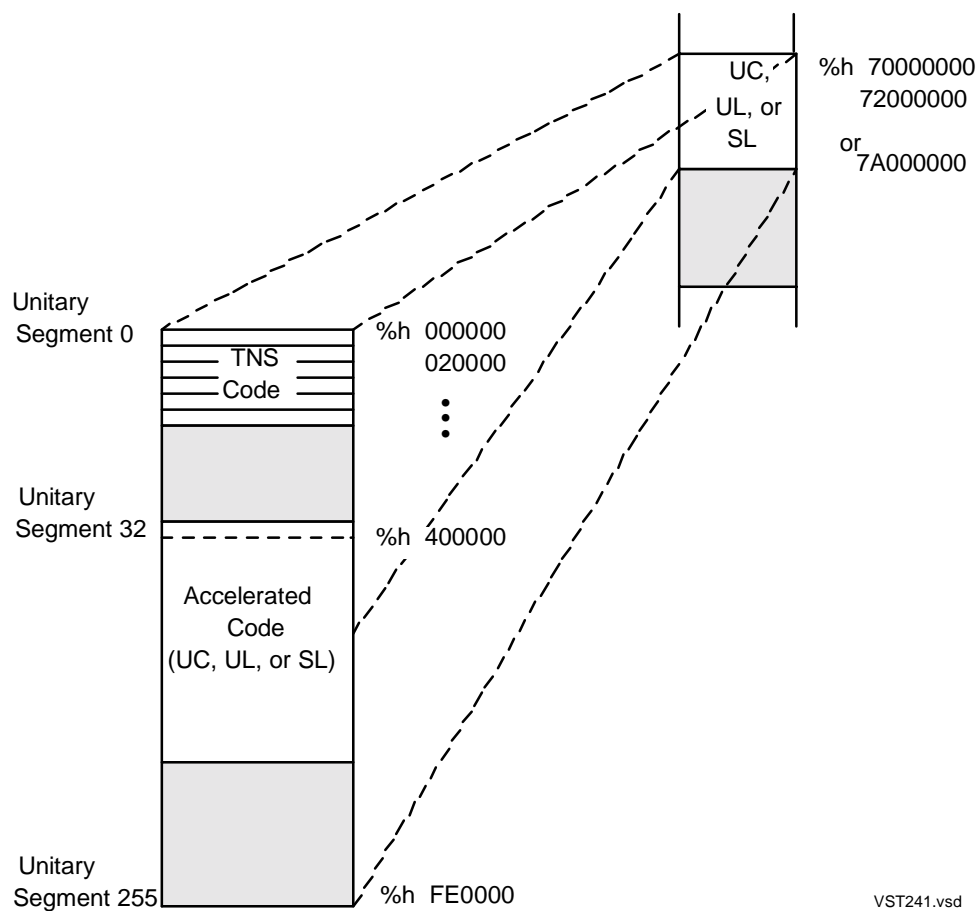
The TNS code for a given TNS process may or may not have been accelerated, depending on the performance desired. However, if the program code has been accelerated, the code region also includes a translation of the TNS object code to functionally equivalent RISC instructions. An accelerated code file will run faster than one that has not been accelerated. That is because only RISC instructions can be directly executed by the instruction processor. A TNS instruction in a nonaccelerated program must be interpreted each time it is encountered at run time.

In either case, the original TNS object code is contained in one or more unitary segments (up to 32) at the beginning of the region, as shown in the figure. At unitary segment 32, only in the case of an accelerated code file, begins the translated RISC code for the region. Whereas the TNS code is broken up by unitary segment boundaries (with external calls between segments), the RISC code is not similarly broken up, and the pieces of code are simply allocated front-to-back throughout the required space, ignoring unitary segment boundaries.

If there is no accelerated code, its absence is indicated by having a page of zeros as the first page in unitary segment 32. Normally, when accelerated code is present, this first page is the beginning of a directory and a set of maps that translate TNS code addresses to corresponding RISC code addresses.

Because RISC code is generally less compact than the original TNS code (plus the need for mapping tables), there is a storage expansion factor for RISC code. This expansion factor can vary considerably, depending on the program and the Accelerator options used; factors of two to six times are fairly typical. Given the amount of space available for the RISC code, a full 32-segment TNS program (an unlikely size) can expand up to seven times. Smaller programs can expand even more—again, an unlikely situation. Therefore, with 224 unitary segments available in the region, there is adequate space to accommodate such expansion.

The TNS system code region (SC) starts at 80000000 and is organized like the other three TNS code regions, with TNS code segments in some of the first 32 unitary segments and accelerator tables and code starting at 80400000. However, this first Kseg0 region is shared with many other things (described in a later topic). Only a few TNS system code segments exist, and the first one is at SC.5, not SC.0.

**Figure 4-11. Both TNS and Accelerated Code Are Included in a Code Region**

# Chart of Nonprivileged Space Allocation

The chart shown in [Table 4-1](#) applies to the G05.00 and G06.00 RVUs. RVUs prior to G05.00 used different space allocations, and future releases can be expected to change again.

For convenient reference, the nonprivileged space is shown divided into 64 regions, each identified with the hexadecimal starting address of the region, abbreviated to the first two digits.

The first region (00) contains the TNS user data, including the TNS stack and global data in the very first unitary segment of the region. This unitary segment is used only in the case of a TNS process. Most of this first region is available for the selectable segment, if one is currently in use. The selectable segment begins at relative segment 4 and can cross one, two, or three region boundaries, depending on its allocated size.

The 35 regions numbered 08 through 4C make up an area that is jointly shared in a special sense: allocations for separate purposes are made at both ends of this space, with no specific address marking the separation. Allocations for native global data (for a native process) begin at the start of the 08 region, proceeding in the increasing address direction. Following those allocations is the native HEAP (for a native process). Allocations for user-allocatable flat segments begin at the opposite end of this shared area (4DFFFFFF), proceeding in the decreasing address direction.

The main stack, which is allocated for all TNS processes and all nonprivileged native processes, begins at the end of the 4E region. Because the main stack grows in the direction of lower addresses, the stack begins one byte before the start of the 50 region; that is, at 4FFFFFFF.

Instance data for a private shared run-time library (SRL), if present, is allocated in the 50 region. Likewise, instance data for public shared run-time libraries is allocated in the 58 and 5A regions. (Intervening regions are reserved for future SRL expansion capability.)

The two regions at 70 and 72 are available for user code and user library. For native processes, the two regions are available for user code (UCr). For TNS processes, one region is available for user code (UC) and one for user library (UL).

Text for a private SRL is allocated in the 74 region, the native user library for a process. Note that there is no “ULr.” Text for public SRLs is allocated in the 76 and 78 regions.

For all processes, the system library (SLr) is addressable in region 7C, and for TNS processes the system library (SL) is addressable in region 7A.

Region 7E contains special millicode, including the TNS code interpreter, all to-RISC gateways, the Debug restart area, the shell map, and the RPWRAP segment.

**Table 4-1. Summary of Nonprivileged Space Allocation**

<b>Region</b>	<b>Allocation</b>	<b>Region</b>	<b>Allocation</b>
00	TNS user data, start selectable seg	40	
02	continue selectable segment	42	
04	continue selectable segment	44	
06	continue selectable segment	46	
08	Global data for native process, followed by HEAP area	48	
0A		4A	
0C		4C	Flat segment space, user allocatable
0E		4E	Main stack (down from 50000000)
10		50	Private SRL instance data
12		52	(Reserved for private SRL data)
14		54	(Reserved for public SRL data)
16		56	(Reserved for public SRL data)
18		58	Public SRL instance data
1A		5A	Public SRL instance data
1C		5C	
1E		5E	
20 - 28		60 - 68	
2A		6A	
2C		6C	
2E		6E	
30		70	UCr or UC (TNS)
32		72	UCr or UL (TNS)
34		74	Native UL, private SRL text, EVs
36		76	Public SRL text
38		78	Public SRL text
3A		7A	SL
3C		7C	SLr
3E	Flat segment space, user allocatable	7E	7E0: Millicode; 7F8: gatewayarea; 7FF0: Debug restart; 7FFC: shell map; 7FFE: RPWRAP, CPUATTR

# Physical Memory Addressing

The preceding topics describe the nonprivileged space. This and the next two topics describe the privileged space.

As indicated earlier, privileged space addresses are signified by a 1 in the most significant bit position of the 32-bit address word.

For segment-numbering purposes, physical addresses can be considered to be a special form of absolute address. Thus the entire privileged space has a range of 16,384 absolute segments, numbered from 0 through 16,383.

Absolute segments 0 through 4095 provide a range of physical addresses to address physical memory through **Kseg0**. This correspondence is illustrated in [Figure 4-12](#). Similarly, absolute segments 4096 through 8191 provide a range of addresses to address physical memory through **Kseg1**. In each case, the addressing range is 512 megabytes.

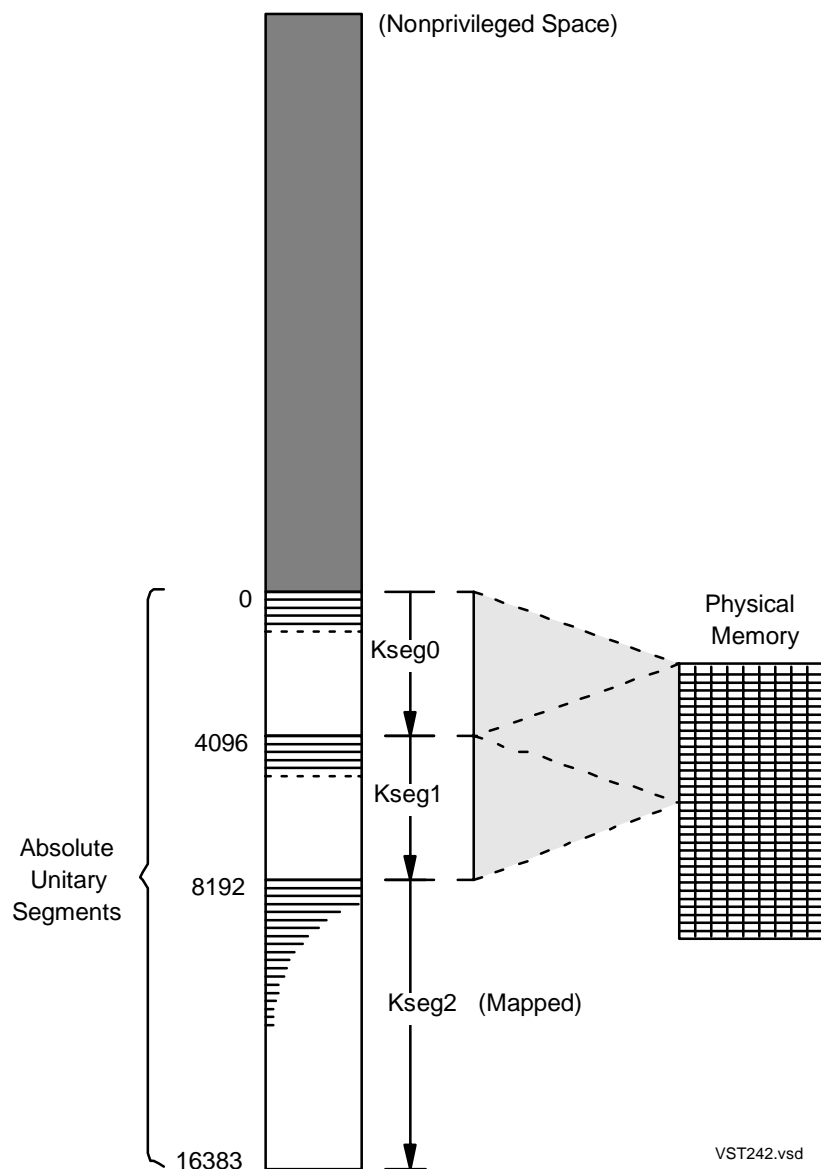
Kseg0 and Kseg1 addresses correspond directly to the physical memory designated by the low-order 29 bits of the address. That is, byte 0 of absolute segment 0 (in Kseg0) and byte 0 of absolute segment 4096 (in Kseg1) both designate byte 0 of physical memory.

The difference between Kseg0 addressing and Kseg1 addressing is that Kseg0 uses the data and instruction caches (the more normal case), whereas Kseg1 does not (and is the more exceptional case). The use of caches is described later, when access techniques are described.

Because installed hardware memory can exceed 512 megabytes, Kseg0 and Kseg1 can have their correspondence with only the first 512 megabytes of installed memory. Access to memory in excess of 512 megabytes can be achieved only by mapping through page tables—that is, through nonprivileged space or Kseg2.

Within the entire 4-gigabyte range of the process address space, the Kseg0 hexadecimal addresses are 80000000 through 9FFFFFFF. For Kseg1, the range is A0000000 through BFFFFFFF.

Actual use of Kseg0 space is described in the next topic.

**Figure 4-12. Kseg0 and Kseg1 Both Map Permanently to Physical Memory**

# Kseg0 Usage

Kseg0 allows cached access to the first 512 megabytes of installed physical memory.

As indicated in [Figure 4-13](#), some unitary segments of Kseg0 are allocated as dedicated, permanently resident parts of physical memory. The remaining space (shown shaded) is available for dynamic memory management. Dedicated allocations include the system data segment and the system code segments. Because the entire privileged space requires privileged access, only those procedures that are RESIDENT and either PRIV or CALLABLE are allocated to Kseg0. Nonprivileged or nonresident procedures are instead allocated in the system library (in the nonprivileged space).

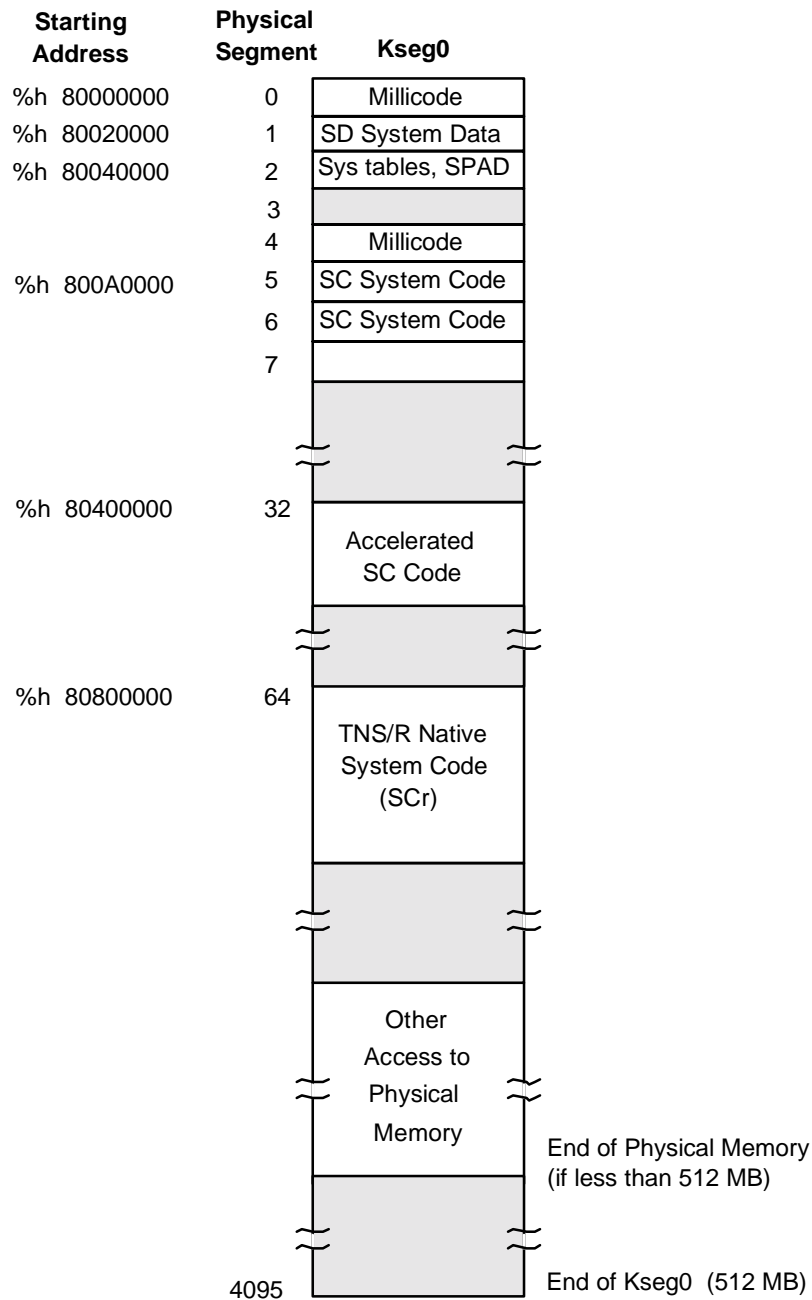
Segment 0 is reserved for low-level millicode, such as the exception handlers, bootstraps, I/O and interprocessor bus handlers, reset initialization, service processor (SP) functions, Lobug functions, and the halt loop. Physical segment 1 is the system data segment, which contains primary global variables for the operating system, including anchor pointers to many system tables; it also includes the memory area called Syspool. Segment 2 also contains various system tables and also the scratchpad (SPAD) frames used by the millicode.

SC is structured as an accelerated TNS code region, as described earlier under [Allocation for TNS and Accelerated Code](#) on page 4-22, but uses only a small part of the first region of Kseg0. There are only a few TNS code segments, starting with SC.5 at 800A0000. The accelerator tables and code begin at 80400000 (absolute segment 32). TNS/R native code (SCr) begins at 80800000.

As evident from [Figure 4-13](#), there is much unassigned address space before, between, and after the system code areas. Although not shown, such space is used by SYSGENR to allocate important resident data structures such as the process control blocks, the address mapping tables, and various other static data structures.

Most of the physical memory (and therefore most of Kseg0) is mapped dynamically for access through nonprivileged space and Kseg2 addresses. A Kseg0 address cannot safely be used to access a page frame that is also mapped to another address; cache coherency errors are likely. Therefore, Kseg0 addressing is restricted to areas, such as those described above, that have no other addresses.



**Figure 4-13. Kseg0 Is Used Primarily for Addressing System Code and Data**

VST243.vsd

# Kseg2 Usage

The **Kseg2** portion of privileged space comprises absolute segments 8192 through 16383. Like nonprivileged space, Kseg2 is organized into regions. Being half the size of nonprivileged space, however, Kseg2 has 32 regions instead of 64.

Most of the Kseg2 space, as shown in [Figure 4-14](#), is used to address absolute memory and is therefore global in nature. However, the third region of this space (region C4) is nonglobal and, for each process, is used for the allocation of its **process file segment (PFS)**, its privileged stack, its **debug stack**, and other temporary information related to the particular process. In addition, the last three segments of Kseg2 are reserved. A portion of this area is kept empty to provide a useful range of nil addresses, which are never mapped to physical memory.

Out of the first two regions in Kseg2, many of the first 128 unitary segments (the first region) are reserved for system allocations. This area includes absolute memory allocations for the system library, system processes created by SYSGENR, and so on. The remainder of those first two regions provide the initial part of the absolute memory allocations for the many processes that exist in the processor: user code, user library, user data, and logical segments. (The remaining, greater, part of the absolute memory allocations is available beginning in the fourth region, C6.)

With the exception of unaliased segments, every process-relative unitary segment in all existing nonprivileged spaces has a corresponding image in absolute memory. Therefore each process-relative address has a corresponding absolute address.

The reason for having absolute addresses in addition to process-relative addresses is to provide a context-independent (global) means of addressing elements belonging to a given process (a context). Such addresses are needed by, for example, interrupt handlers. The process-relative address alone is not meaningful to these routines. Absolute addressing provides an **alias** as a means of accessing process elements. Any segment that also has an absolute address is said to be a **kernel-aliased segment**. The TNS user data segment and all code segments have associated absolute segments; the RISC stacks, heap, native globals, and SRL instance data do not. Therefore user code (UC), user library (UL), system code (SC), and SRL code are all aliased. By default, both flat segments and selectable segments are unaliased; however, the application must be a privileged program to have unaliased versions of segments.

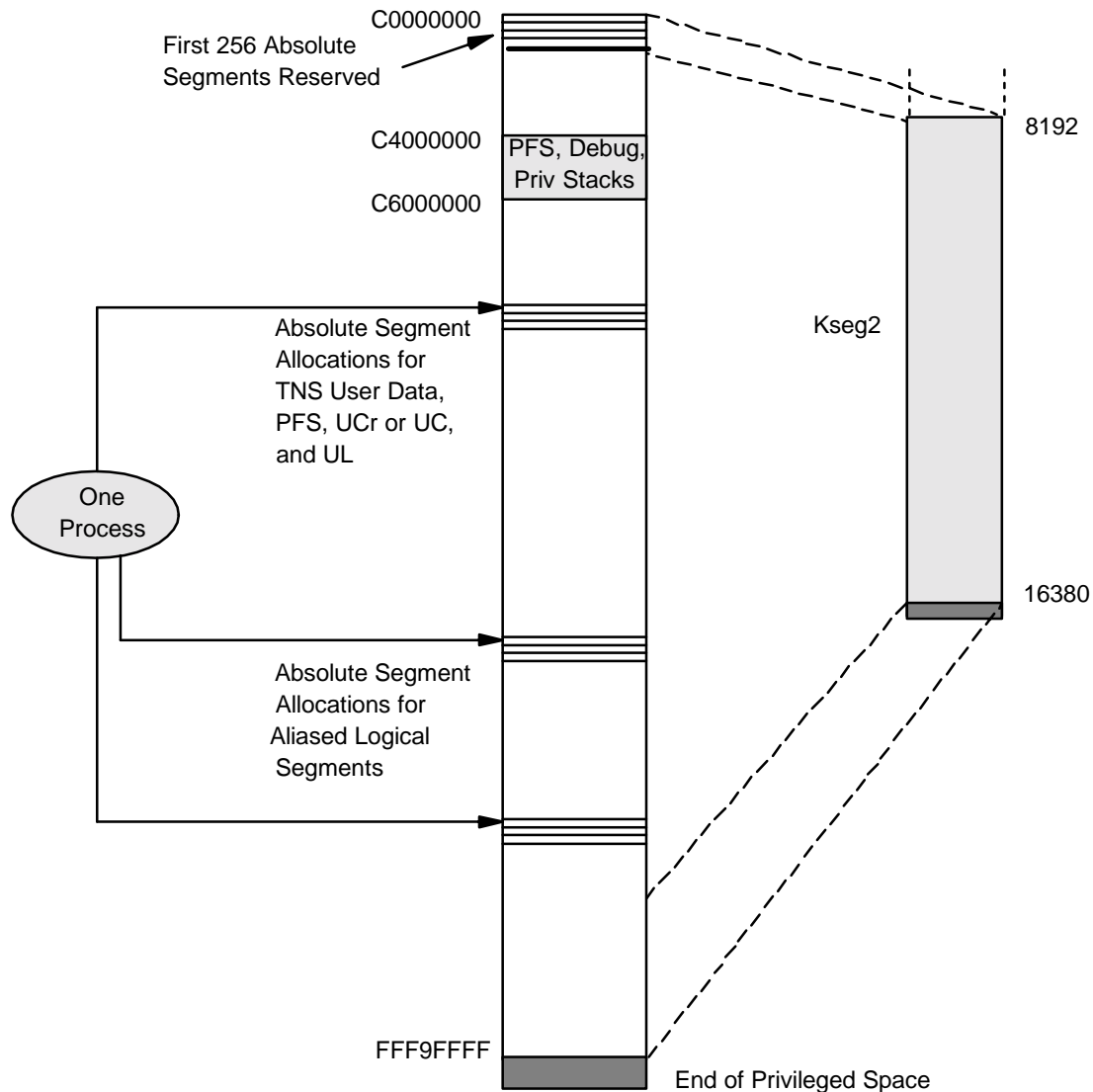
Another means of accessing process elements (context-bound addresses for unaliased segments) is discussed under [Context-Bound Addresses](#) on page 4-50.

Each kernel-aliased logical segment (code or data) has a corresponding set of consecutive absolute segments. For TNS processes, there is one absolute segment for user data.

The absolute segments constituting a single logical segment are contiguous, but the various segments of a process have unrelated absolute addresses. The absolute addresses of an aliased logical segment change if the segment is enlarged with

RESIZESEGMENT. There can be up to 1024 absolute segments (or 128 megabytes) for each logical segment that might exist for the process.

**Figure 4-14. Kseg2 Provides Absolute Memory Allocations for Every Process**



VST244.vsd

# Kseg1 Memory Access

The preceding topics in this section explain the concepts of virtual memory addressing. This and the remaining topics in this section consider how physical memory is actually accessed in the processor. Because access to memory through Kseg1 is the simplest, it is described first, illustrated in [Figure 4-15](#).

The memory caches are not used when memory is addressed through Kseg1. All such memory references go directly to physical memory and are used for memory-mapped hardware addressing. As shown by the numbered steps in the figure, the virtual address from the processor is taken as the physical address (simply by stripping the three most significant bits of the 32-bit address) and applied directly to the memory hardware for access.

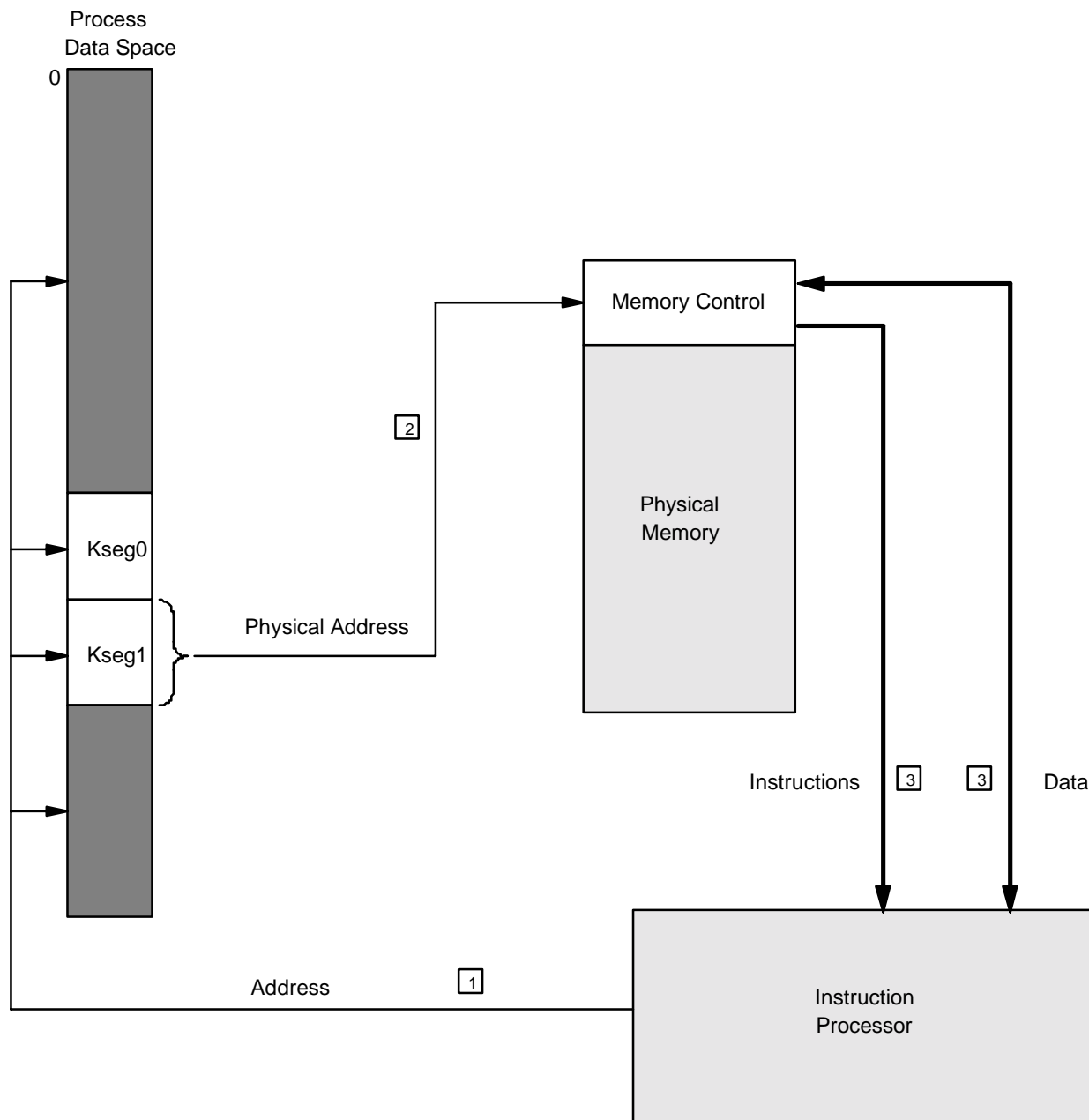
Although this mode of access is simple and direct, it does have negative considerations. The first involves, of course, the reason why memory caches were invented in the first place—to enhance performance for frequently accessed code and data. For example, load instructions using uncached addressing require the instruction processor to stall for many cycles until the data is ready.

Another negative consideration is the possibility of cache consistency problems, wherein the same area of memory might be accessed both through the caches and bypassing the caches.

Also, the processor accepts Kseg1 addresses only to access a properly-aligned 32-bit word. Attempting a half-word (16-bit) or byte access via a Kseg1 address causes a hardware-error freeze.

For these reasons, Kseg1 addressing is rarely used in NonStop servers.

The sequence for [Figure 4-15](#) is as follows. (1) The RISC instruction processor applies a virtual address to any of the four areas of process address space. (2) If the address is for Kseg1, it is a physical address and is delivered directly to the memory control logic on the memory boards. (3) The memory control logic accesses physical memory to read or write data or to fetch instructions.

**Figure 4-15. Access Through Kseg1 Is Direct: No Caching, No Address Translation**

VST245.vsd

# Kseg0 Memory Access

Kseg0 addressing is similar to Kseg1 addressing, providing access to exactly the same physical memory. However, memory information is transferred between memory and the **memory caches** in blocks and as individual four-byte words between the caches and the instruction processor. Thus the access is not direct, as in the case of Kseg1 addressing.

There are two levels of caches, designated primary and secondary. Also, there are two primary caches, one for instructions (read only) and one for data (read and write). These are shown in [Figure 4-16](#). The primary caches are comparatively small (16K bytes); they are indexed with the virtual address. The single secondary cache is larger (varies in size depending on processor model; typically a half megabyte, one megabyte, or two megabytes); it is indexed with the physical address. For Kseg0 addresses, the physical address is derived by simply stripping high-order bits from the virtual address.

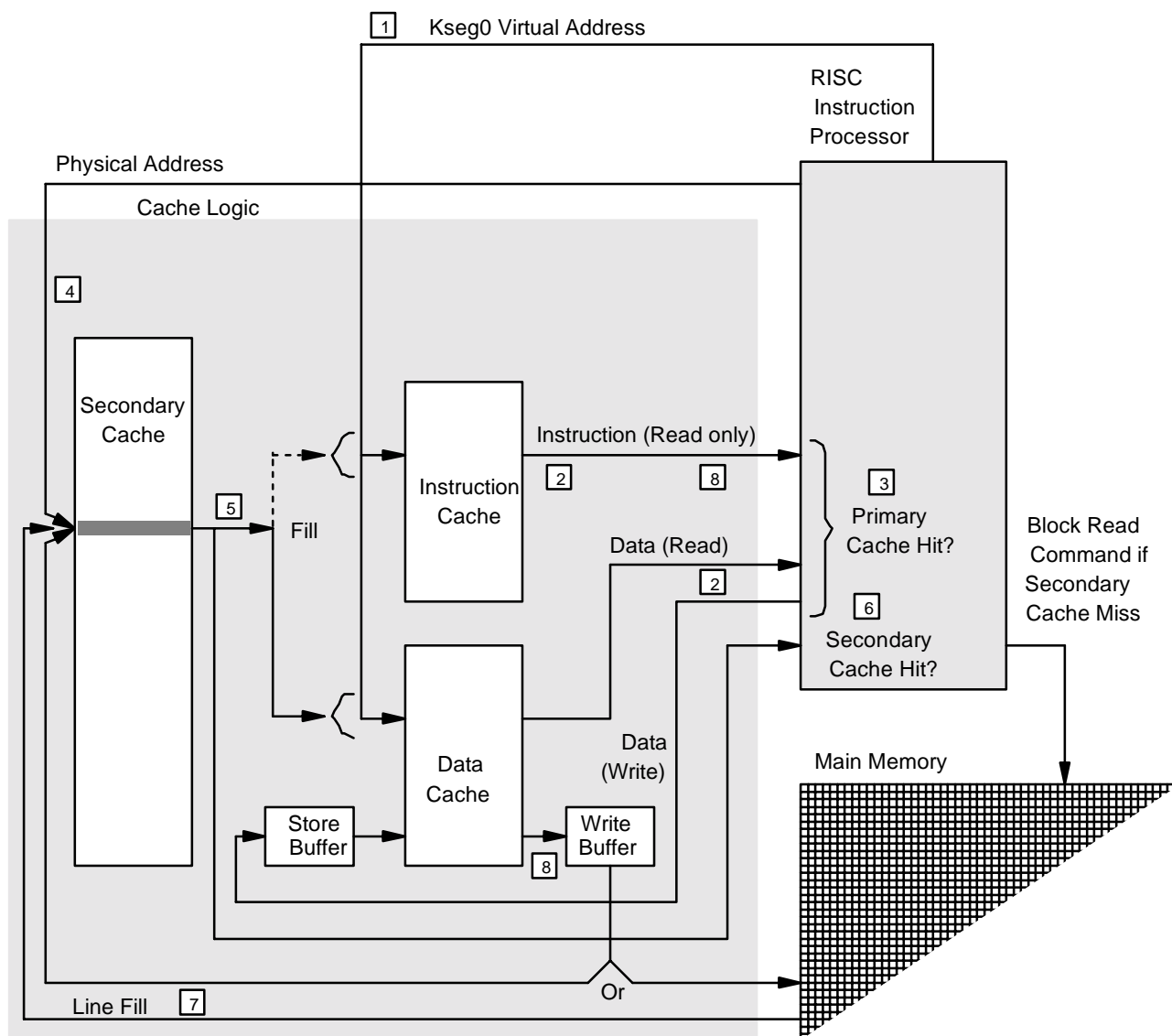
In operation, the RISC instruction processor attempts to read an instruction from (or read data from or write data to) the appropriate primary cache. To do so, it checks the cache tags (physical address part of cache lines) to determine whether one line contains the addressed location. If so, the data word or instruction is read from the instruction or data cache (or is stored into the data cache through a store buffer in the case of writing data). That is a primary **cache hit**.

If the content of the addressed location is not in the primary cache, the instruction processor indexes into the secondary cache using the physical address. It checks the lines contained in the cache for a secondary cache hit. If the line is present in the secondary cache, it is transferred (filled) into the primary cache. Then the read or write can take place with the primary cache.

However, if the content of the addressed location is not in the secondary cache, the processor issues a command to main memory to read a block of data into the secondary cache. At that point, the line is transferred to the primary cache and the addressed information is transferred to or from the instruction processor, in accordance with the original read or write request.

For any write operations, the data is written into cache, merging with the existing data, and at some later time is written to main memory—"write-back" cache.

In summary the sequence is as follows. (1) The RISC instruction processor indexes into the primary caches. (2) For a read, it reads the appropriate cache slot (expecting data or instruction); for a write, it stores data in the data cache through the store buffer. (3) If the data or instruction is in the cache, go to Step 8. (4) On a primary cache miss, the processor indexes into the secondary cache. (5) Line-fill the primary cache. (6) In case of a secondary cache hit, return to Step 2. (7) In case of a secondary cache miss, the processor goes to memory to fill the cache line; return to Step 5. (8) The processor uses the read data or sends write data to memory through the write buffer.

**Figure 4-16. Memory Access Through Kseg0 Uses Memory Caches**

VST246.vsd

# Kseg2 and Nonprivileged Space Memory Access

For Kseg0 addressing (previous topic), the physical address needed for cache access is derived simply by stripping high-order bits from the virtual address. For Kseg2 and nonprivileged space addressing, virtual addresses must specifically be translated to physical.

The RISC processor includes a special-purpose cache called the **translation lookaside buffer (TLB)** to translate virtual addresses to physical addresses. The TLB performs its translations for both forms of virtual address: relative (in the case of nonprivileged space addresses) and absolute (in the case of Kseg2 addresses). The unit of translation is the page; that is, virtual page numbers are translated to physical page numbers (frames).

The TLB can contain only 96 page translations at a time (48 even-odd pairs). Therefore, there is a backup structure in memory, consisting of page mapping tables (detailed later) that are the ultimate source of translations used in the TLB.

In operation, as presented in [Figure 4-17](#), the TLB is scanned (simultaneously and in parallel with the indexing of the primary caches) for a matching virtual page address. A successful match either must have the correct nonprivileged space number or the TLB entry must be marked as “global.” If a match is found, the physical frame number contained in that location of the TLB is the desired translation. That is a **TLB hit**. The cache logic can continue its access, now having a verified physical address.

If no match is found, or if a match is found but the TLB entry is not marked as “valid,” a **TLB miss** has occurred. A succession of translation tables in memory is accessed to find the page location. If the page is in memory, its address is installed in the TLB (this takes a few dozen machine cycles). Otherwise, this is a **page fault**; the desired virtual page is on disk (or nonexistent). The current process gets suspended at this point until the operating system has fetched that page into memory. Then, when that process executes again, the instruction that received the page fault is tried again.

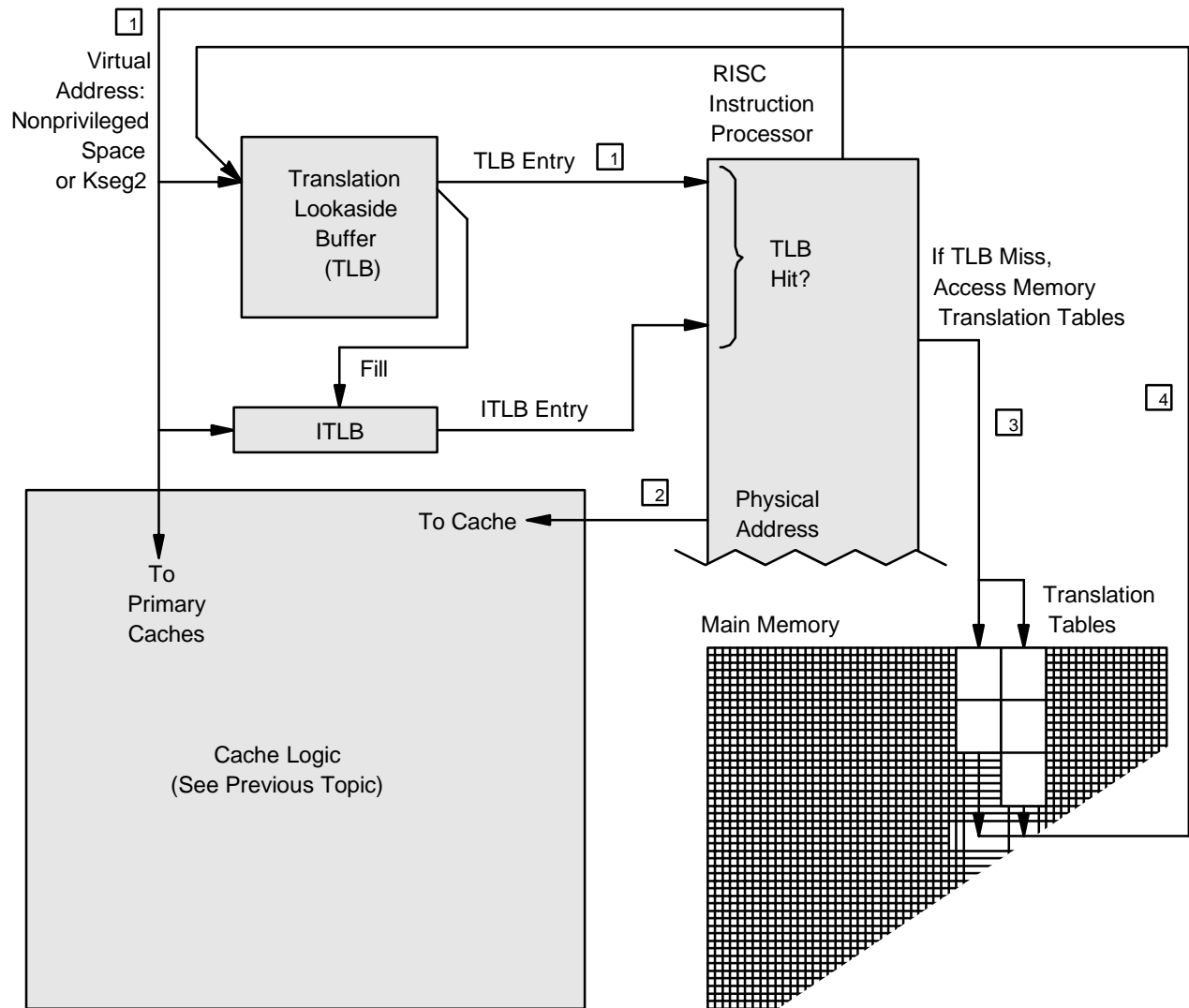
Most of the TLB slots are thus filled by the action of memory reference misses randomly overwriting previous page translations. There is no checking for frequency of reference or other factors for those slots. However, the first few slots of the TLB (specifiable) are reserved by the millicode and are never overwritten.

The instruction translation lookaside buffer (ITLB) is a two-entry substructure that permits instruction address translation to be done in parallel with data address translation. It is filled from the TLB when there is an instruction address miss.

The sequence for [Figure 4-17](#) is as follows. (1) The TLB and ITLB are scanned for a matching virtual address, in parallel with cache indexing. (2) If TLB hit occurs, the physical address is thus verified for cache-hit testing; see previous topic. (3) If TLB miss occurs, translation tables are accessed in memory to compute the physical address. (4) When the address is obtained, it is put in the TLB, and the operation repeats Steps 1 and 2.



**Figure 4-17. Memory Access Through Nonprivileged Space or Kseg2 Requires Address Translation**



VST247.vsd

# The TLBPID Process Identifier

One of the fields in a TLB entry is an 8-bit process identifier. This field makes it possible to distinguish translations that belong to one process from those that belong to another process. (Remember from the very first topic in this section that all processes use the same numerical range of virtual addresses.)

This field is designated as the **TLBPID** (translation lookaside buffer's process identifier), or ASID in RISC terminology (address space identifier). This can be considered a prefix to the address, as shown in [Figure 4-18](#). (A nonprivileged address is shown as an example format.) The instruction processor automatically supplies the current process's TLBPID on every memory reference.

However, because the operating system permits more than 256 processes, the 8-bit field is insufficient for unique identification. Therefore, the processor uses the scheme shown in [Figure 4-18](#) to permit sharing of the 256 TLBPIDs among many processes.

The scheme used for the sharing of TLBPIDs is based on a data structure in physical memory called the **TLBPID owner array**. The array consists of 256 slots corresponding to the 256 possible TLBPIDs.

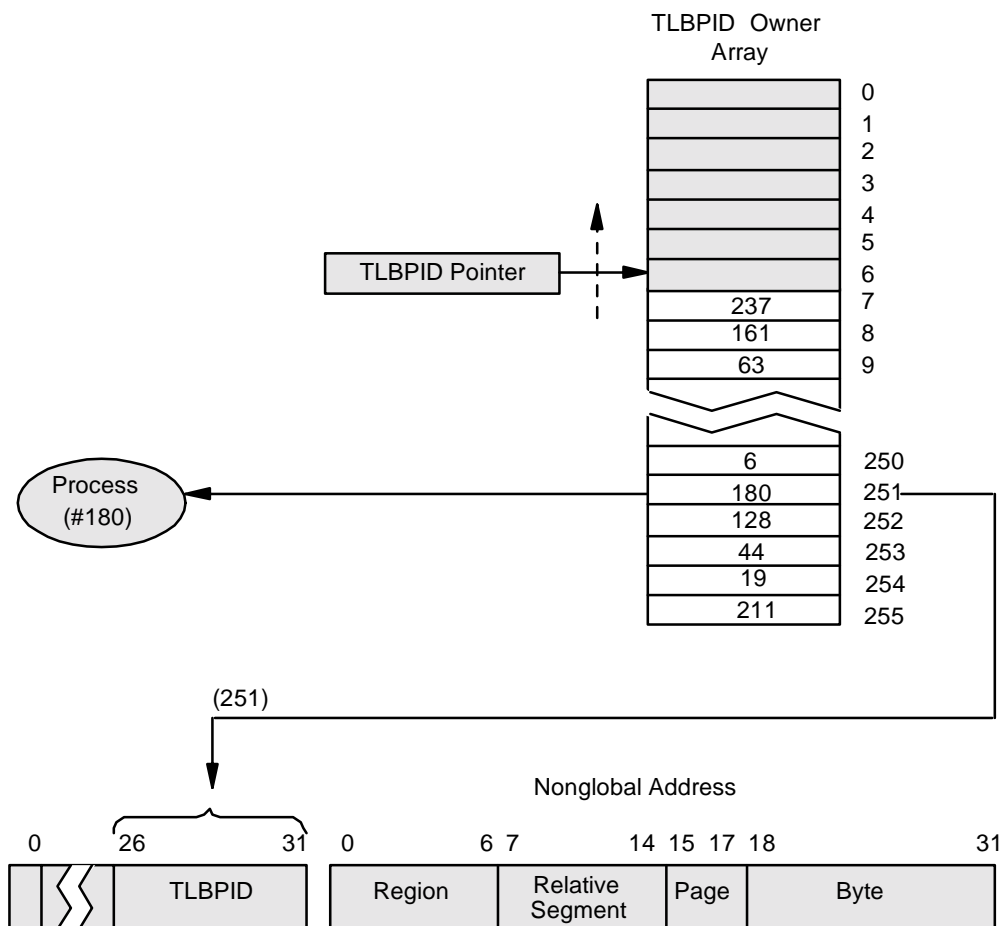
If a process does not currently have a valid TLBPID, the millicode assigns one from the TLBPID owner array. Each entry is a pointer to a location in a table (the process data space table, or PDST, described later) where the corresponding TLBPID number is stored.

The figure shows a simplified example of this assignment method. In this example, TLBPID number 251 is assigned to a process identified here as process number 180. A pointer (represented here by the number 180) to the process data space table (PDST) for that process is entered in the TLBPID owner array, and the number 251 is stored in the process's PDST. The process can then use the TLBPID number 251 in its nonprivileged space references.

A process can have only one TLBPID at a time.

TLBPIDs are assigned on request from processes in descending sequential order, as indicated by a TLBPID pointer. After the last TLBPID is used (0), the entire array is cleared, the random entries of the TLB itself are flushed (marked invalid), and assignments start all over at number 255.

A process keeps its most recent TLBPID assignment until the owner array is cleared and TLB invalidated. The process is then reassigned a new (probably different) TLBPID when it next resumes execution.

**Figure 4-18. The TLBPID Distinguishes the Address Space of a Process**

VST248.vsd

# Nonglobal Address Translation

The entries in the translation lookaside buffer (TLB) are 128 bits wide, as shown in the lower part of [Figure 4-19](#). The high-order 64 bits contain the fields used to identify the page address that is translated in one given slot of the TLB: a **virtual page number (VPN)** and a translation lookaside buffer's process identifier (TLBPID). Any given slot of the TLB has two translations: one for an even virtual address and one for the immediately following odd virtual address. These two translations occur in the low-order 64 bits of the TLB entry.

The mask field specifies a page size, which is 16,384 bytes in the random TLB entries.

The TLBPID is described in the preceding topic. It specifies a process address space. The VPN term is RISC terminology and is directly equivalent, in NonStop S-series processor terms, to all of the most significant fields of a virtual address down to and including the page field. Generally, this is an 18-bit field, bits 0 through 17 of a virtual address; however, because of the even-odd pairing of translations (used for increased efficiency), only 17 bits are used for comparison—that is, VPN divided by two, shown as VPN/2. The low-order bit of the page field in the virtual address is used as an even-odd selector in selecting one of the two translations in the TLB slot.

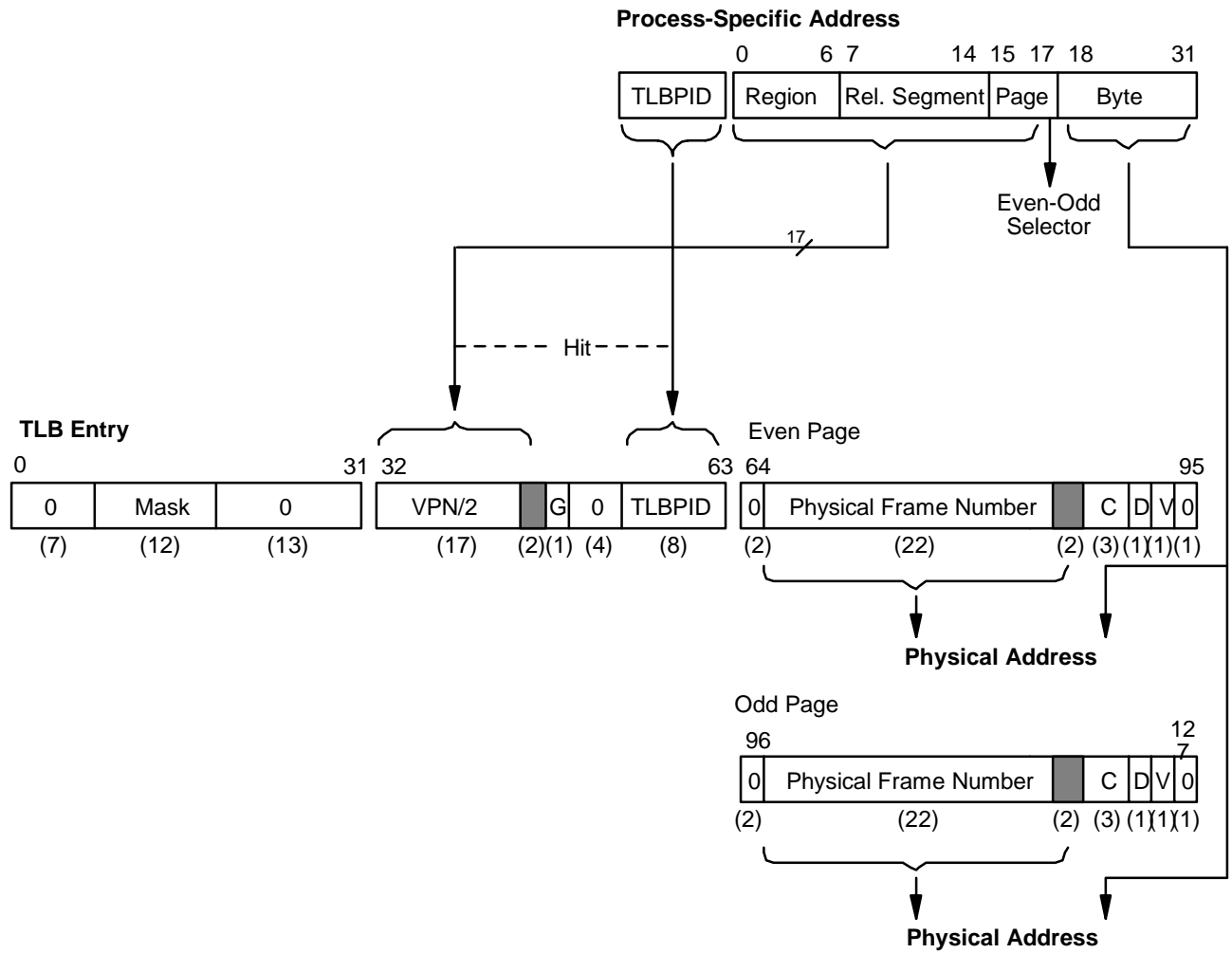
Nonglobal address translations need both of these fields in checking for a translation hit. Addresses using global addressing do not require a matching TLBPID; translation hits for such global elements are described in the next topic. The following paragraphs describe the nonglobal case (specified by the one-bit G field being 0).

When a nonglobal address is applied to the TLB for translation, the TLB logic scans the combination of VPN/2 and TLBPID fields, looking for a match with the combination of the supplied TLBPID of the process and the first 18 bits of the address (ignoring the lowest-order bit). In a nonglobal address, the first 18 bits comprise four fields: the relative/absolute identifier, the region number, the relative segment number, and the page number.

When an exact match is found in one of the 48 slots of the TLB, a translation hit has occurred. The 22-bit field in bits 65 through 87 of the TLB entry (or 97 through 119 in the case of an odd page) is then taken as the frame number of the target physical address. The frame number field combined with the byte field of the nonglobal address produces a 32-bit physical address that is capable of addressing a 4-gigabyte range of physical memory.

TLB entry bits C, D, and V are used for special purposes and are described in the succeeding topics.

In summary, referring to [Figure 4-19](#): together, the first four (high-order) fields of the nonglobal address form what the RISC processor considers to be the virtual page number. In effect, the TLBPID is a superior part of the address. These two fields are compared with corresponding fields in TLB entries. For a matching entry, the translated frame number is in the 22-bit field from bits 65 through 87 (even page) or 97 through 119 (odd page).

**Figure 4-19. Nonglobal Address Translations Require Matching TLBPID**

# Address Translation of Global Elements

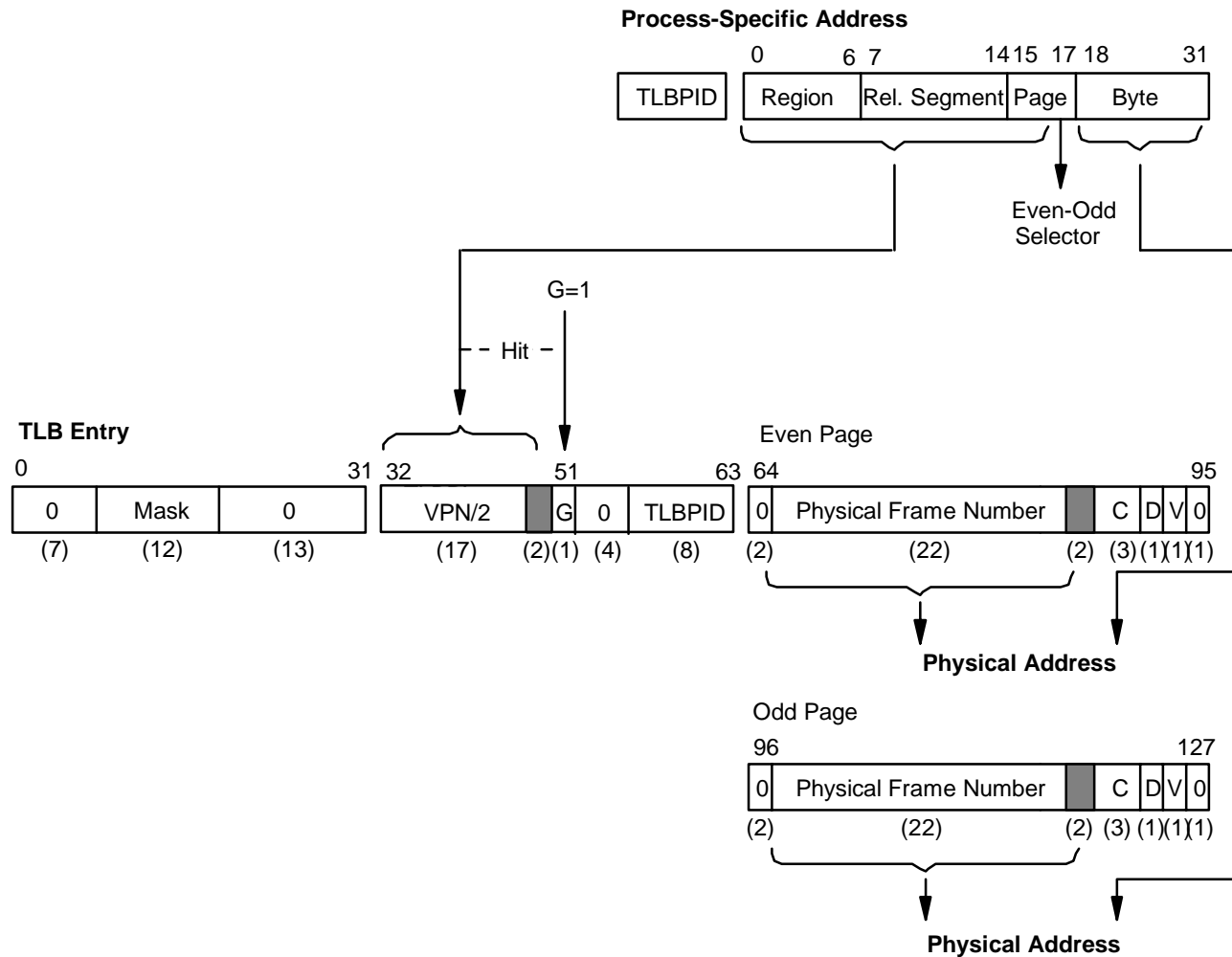
As was shown in the first topic of this section, parts of the nonprivileged space and most of the privileged space of virtual memory are global among the currently defined processes. The entire range of virtual addresses constituting absolute virtual memory (most of Kseg2), for example, is global to all process address spaces. Some parts of nonprivileged space, such as the system library and RISC millicode regions, also need to be global to all process address spaces.

In all such cases, no process identification (TLBPID) is needed. Instead, for any global addresses entered in the TLB, the Global bit (bit 51) is set to 1. When the TLB is scanned for matching virtual addresses, the TLB logic ignores the TLBPID field if the Global bit is equal to 1. In that case, a translation hit occurs whenever the virtual page number field of the TLB entry matches the 17 most significant bits of the virtual address. (As mentioned in the previous topic, the 18th bit is an even-odd selector and is ignored when scanning for a hit.)

[Figure 4-20](#) illustrates address translation for a global address in either the nonprivileged space or the Kseg2 space.

Although the RISC chips provide for 64-bit addressing, and consequently an alternate TLB format, this capability is not used in the NonStop S-series processors.

The 17 most significant bits of any virtual address are compared with the virtual page number field of the TLB entry. If these fields match, and if the Global bit is set to 1, a translation hit is achieved. The TLBPID field of the TLB entry is ignored. The Global bit is set to 1 for all absolute segment pages and any other pages that are shared among all processes.

**Figure 4-20. Address Translation of Global Elements Ignores TLBPID**

VST250.vsd

# Address-Mapping Tables

The translation lookaside buffer (TLB) is located on the RISC chip, and it is extremely fast in performing virtual-to-physical address translations—actually in half a machine cycle. However, any time a reference is made to a location in a page that does not currently have a translation in the TLB (a translation miss), the memory-management millicode must obtain the page translation from memory tables. Once that translation is found (as part of a pair), the pair is stored in the TLB, overlaying some existing pair of translations. The desired translation is then used for the current reference and possibly future references, until it is itself replaced.

[Figure 4-21](#) shows how a pair of translations is obtained from the memory tables, for the case of nonprivileged and Kseg2 space addresses. All of these tables are in Kseg0 memory, so they are resident and can be accessed without using the TLB. Note that the objective is to store a pair of physical frame numbers in the right half of a TLB entry.

As a first step, the region number in the region field of the address is used to index into the process data space table (PDST) that is allocated for this process. The location pointed to contains a 32-bit pointer to the beginning of one of 128 possible **Vseg tables**. One Vseg table exists for each region that is currently allocated to the nonprivileged space and the Kseg2 space. Note that region numbers 64 through 95 are not used, because they indicate Kseg0 and Kseg1; these and some other PDST entries point to the null Vseg, described later.

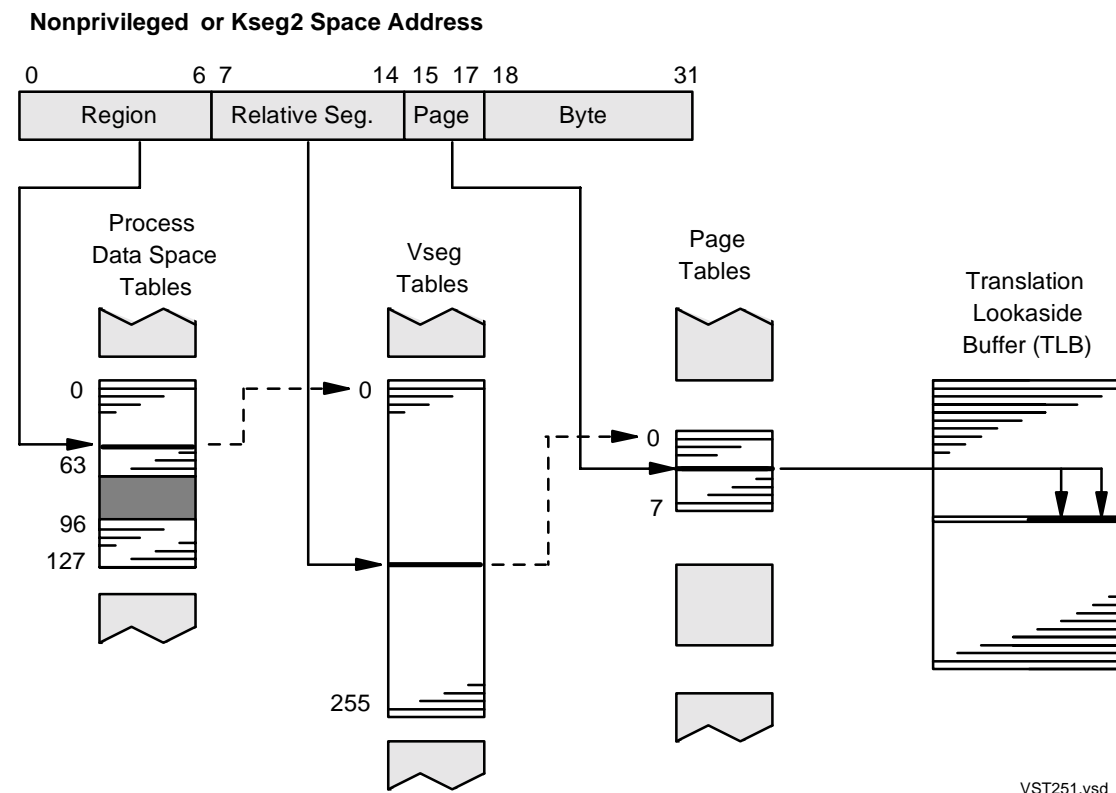
Next, the relative segment number in the relative segment field of the address is used to index into the selected Vseg table. The location pointed to contains another 32-bit pointer, which points to the beginning of one of 256 **page tables (PTs)**. One PT exists for each segment that is currently allocated to the region. (Other Vseg entries point to the null PT, described later.)

Finally, the page number in the page field of the address is used to index into the selected page table. The location pointed to (if the V bit indicates valid) contains the frame number needed to complete the translation. This content and the one in the location preceding (if odd page) or following (if even page), are then loaded into the right half of one slot in the TLB, along with the virtual page number in the left half. These translations are now available for the current virtual reference.

If the V bit does not indicate valid (that is, the page is absent from physical memory), a page-fault interrupt invokes the memory manager software to load the page from disk to memory and to set up the page table entry. Then, when the process next executes, the instruction that caused the TLB-miss exception is tried again.



**Figure 4-21. On TLB Miss, Nonprivileged Space and Kseg2 Space Translation Uses Address-Mapping Tables**



# Access of Special Pages

Some special processor features need to be implemented as reserved virtual memory spaces. Three specific features are described in this topic: (1) a memory-resident TNS register stack, located in a page called the **RP wrap page**, (2) a range of nil addresses, and (3) a scratchpad for millicode use, called the **SPAD pages**.

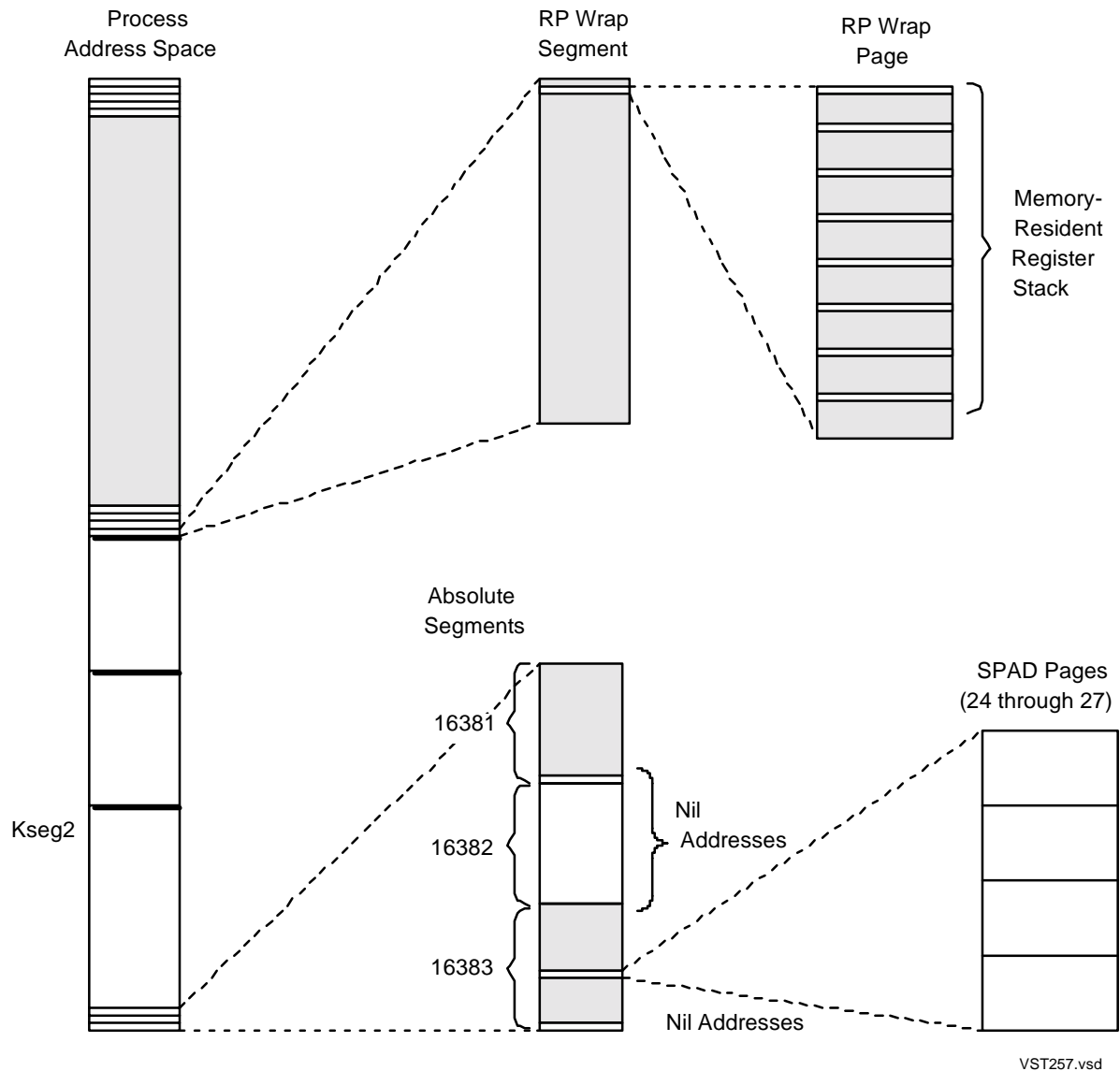
The simulated register stack is allocated in the last unitary segment of the nonprivileged space. As shown in [Figure 4-22](#), the memory-resident TNS register stack (described in [Section 6, TNS Execution Modes](#)) is located in a single page called the RP (for register pointer) wrap page. This page is the second page (page 1) of a segment called the RP wrap segment. The hexadecimal address of the page is 7FFE4000. Other pages of the segment are all unused.

The RP wrap page is permanently resident in physical memory. Although it is located in nonprivileged space, it is accessed only by TNS interpreter millicode, and callable procedures reject attempts to use RP wrap locations as reference parameters.

As for nil addresses, these are needed by the operating system to guarantee that specific ranges of addresses result in memory address error traps; thus there will never be virtual memory allocated at these addresses. These ranges are specified as –256 KB through –128 KB –1, and –2 KB through –1. These ranges fall at the end of Kseg2 and represent the last two kilobytes of absolute segment 16381, all of absolute segment 16382, and the last two kilobytes of absolute segment 16383. The operating system never allocates virtual memory in any of these three segments.

The SPAD (scratchpad) is a group of four pages allocated in absolute segment 16383, specifically pages 24 through 27 in the full range of 0 through 31. These pages actually do get translated to physical memory, and they allow privileged millicode to access memory variables quickly and without disturbing the contents of the general registers.

The scratchpad pages contain many privileged-only data items that need to be accessed by millicode. Those related to memory management include the TLBPID owner array, pointers to the PDST and SEG tables, and a pointer to the current process data space table (CPDST). The SPAD pages are allocated, locked, and initialized by SYSGEN, and their address translations are permanently loaded into reserved locations of the translation lookaside buffer (TLB).

**Figure 4-22. Special Pages Are Accessed Through Fixed Addresses**

## Defining Unallocated Space

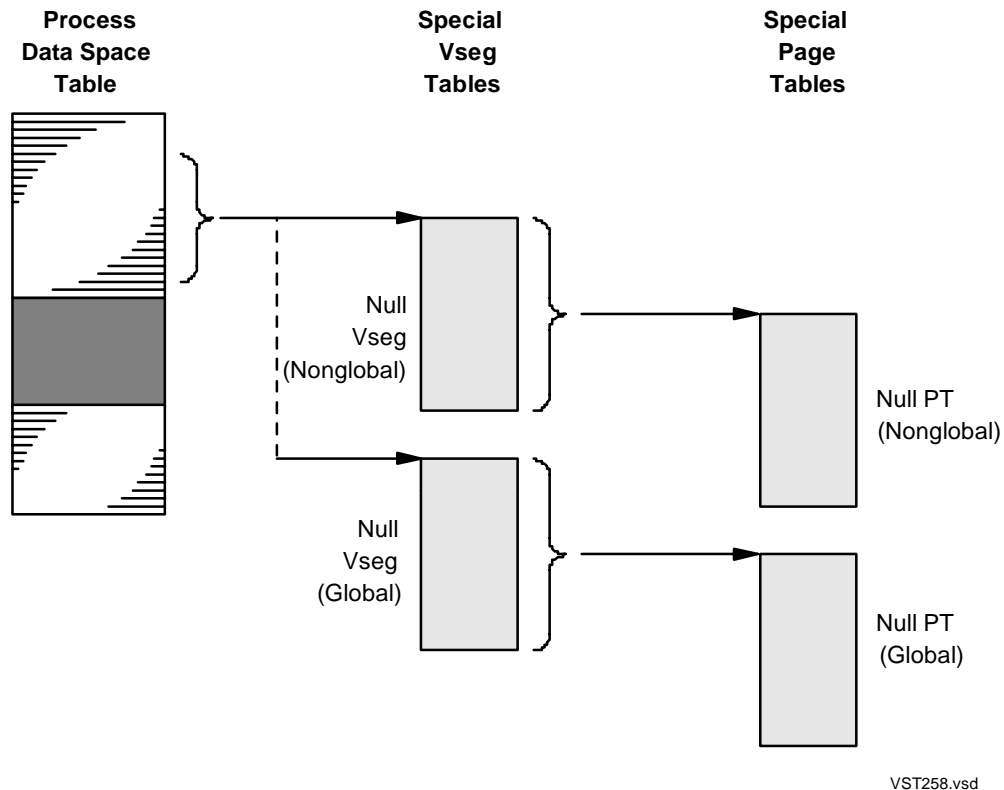
Whereas translation misses in the translation lookaside buffer (TLB) are quite common, invalid addresses (those that point to nonexistent memory) are rare. It is reasonable therefore to eliminate explicit validity checking of addresses, such as checking against a limit, and instead handle invalid addresses as exceptional cases. Thus the TLB-miss millicode can be very fast, whereas address errors can be allowed to suffer the consequences of special handling.

In the NonStop S-series processor, the method for handling invalid addresses is to have all table entries for invalid addresses point to a table entry that permanently specifies that the page is not valid (absent). The page fault invokes a trap handler because no memory exists. [Figure 4-23](#) shows how this scheme is implemented for addresses in the nonprivileged space or in the Kseg2 space.

Each processor contains two null page tables (PTs), one for nonglobal addressing and one for global addressing. These tables are allocated during system initialization in physical memory. These PTs have the Valid (V) bit permanently set to 0 in all 8 entries. Thus any reference to a page in these PTs results in a page fault trap.

When Vseg tables (in the case of either nonprivileged space or Kseg2) are allocated, all unassigned unitary segment entries are made to point to one of these two null PTs. That action ensures that references to unassigned relative or absolute segments will trap.

In many cases, an entire region is unassigned. To avoid the necessity of many entire Vseg tables pointing to the null PT, pointers in the process data space table (PDST) point to one of two null Vseg tables (one for nonglobal, one for global), which in turn have all their 256 entries point to the corresponding null PT.

**Figure 4-23. Unallocated Space Is Defined With a Few Null Tables**

# Context-Bound Addresses

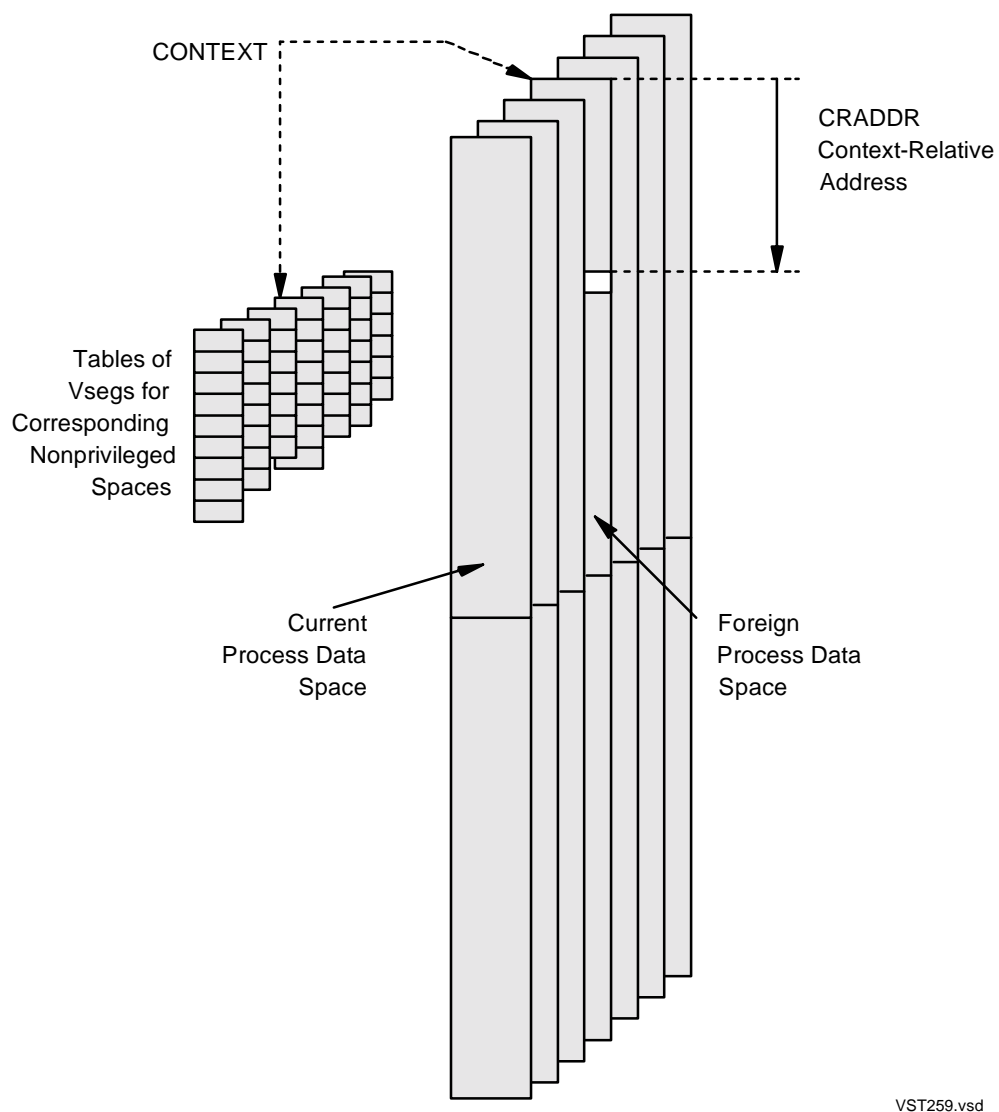
Because only one addressing space, Kseg2, is available for providing aliases for all the existing nonprivileged spaces (one or more per process), absolute addressing as an alias has a finite limitation. Therefore, an alternate means of aliasing is provided, called **context-bound addressing**. This is a software facility. It permits logical segments to be created without absolute-address aliases. Such logical segments are known as **unaliased segments** and are specified with an option parameter to the `SEGMENT_ALLOCATE_` procedure. Context-bound addresses (CBAs) can represent any location in all the process address spaces. CBAs can be used only in privileged state.

As an example of use, an operating system process might be the current process address space; the process address space that provides a certain service for this process is in a **foreign process address space**, as illustrated in [Figure 4-24](#). The operating system process uses context-bound addresses to access the foreign process address space, which performs the requested service.

A CBA consists of two separate components: a `CONTEXT` specifier and a context-relative address (`CRADDR`). `CONTEXT` is hidden from the program, but it is implemented as a low-level table address. `CRADDR` is a relative address, the offset from the start of the nonprivileged space.

For all processors, `CONTEXT` is the starting address of an array of pointers to Vseg tables for the foreign address space. For flat segments, this array is the region table in the PDST. For selectable segments, this array is an appendix to the operating system structure for that logical segment. For some situations, a degenerate CBA is used, with a zero `CONTEXT` and an absolute (Kseg2) address in `CRADDR`.

**Figure 4-24. Context-Bound Addresses Substitute for Aliases in Unaliased Segments**







# Instruction Processing Environments

This section describes the difference between native and TNS processes, their stacks and segment allocations, as well as mode transitions that occur during the execution of different types of processes. The topics covered in this section are:

[Native and TNS Programs and Processes](#)

[Stacks for Native and TNS Processes](#)

[Code and Data Allocations](#)

[Sharing of Code and Data Segments](#)

[Restricting Mode Transitions](#)

# Native and TNS Programs and Processes

The RISC processor that is the instruction processing unit (IPU) of each NonStop S-series processor executes only RISC instructions. However, to accommodate the large installed base of TNS programs, the processor has been designed to accept and execute TNS programs in addition to programs that are compiled directly to RISC instructions.

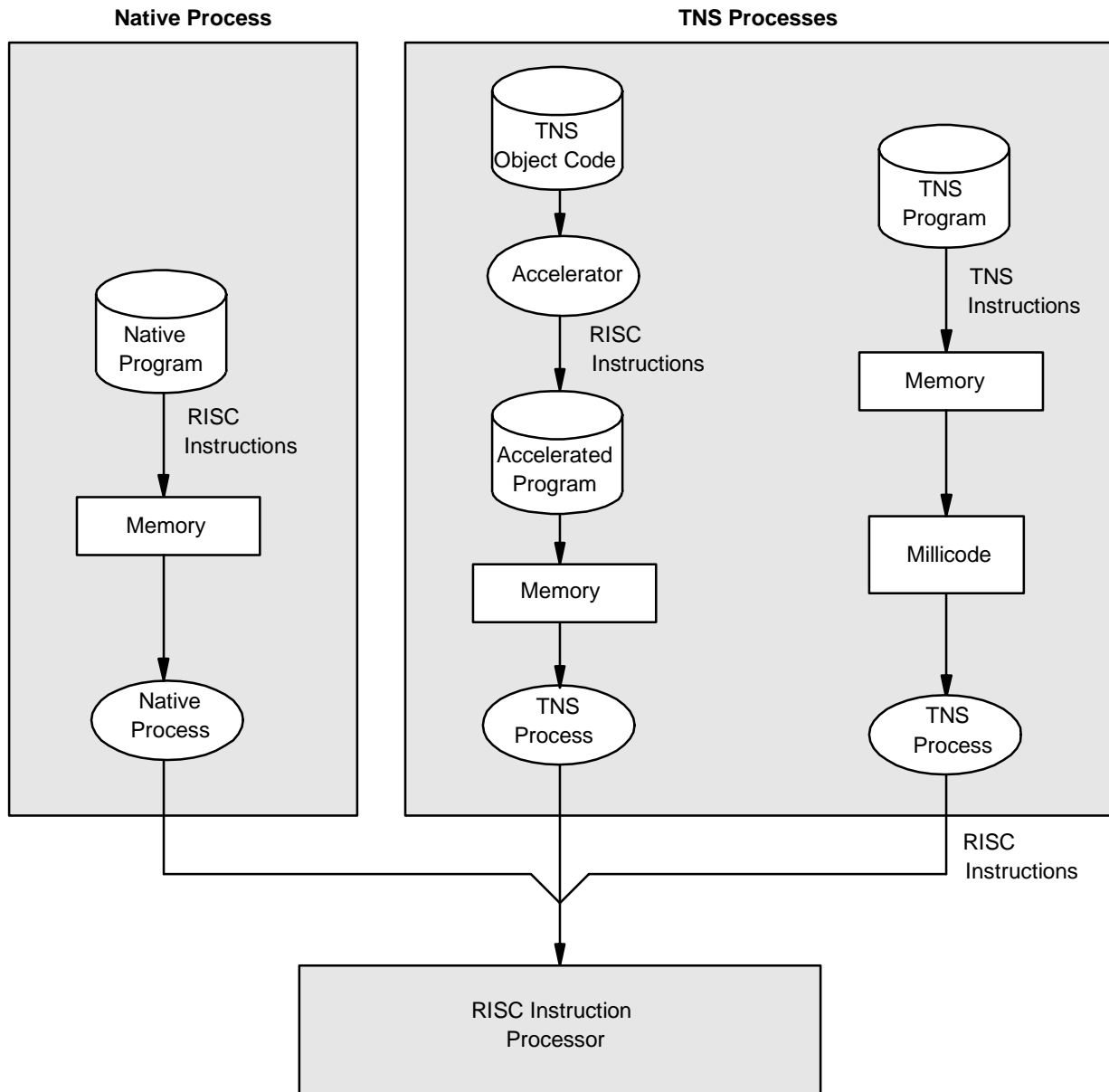
Applications that are compiled directly to RISC instructions are in the form of **native object code**. Such code, when executed, becomes a **native process**, illustrated in the leftmost sequence in [Figure 5-1](#). The RISC processor reads RISC instructions from memory (backed up by the object code file) and executes those instructions directly, using RISC stack conventions. RISC stack conventions are described in [Section 7, Native Execution Mode](#).

Applications that have been compiled as TNS instructions require the TNS instructions to be converted to RISC instructions. There are two methods of performing this conversion, differing in how and when the instruction conversion is performed.

In the case of an **accelerated program** (middle sequence in [Figure 5-1](#)), the TNS code is converted to RISC instructions by an Accelerator program prior to run time. The result of this preprocessing is that accelerated code (translated sequences of RISC instructions) exists in the same code file as the original TNS object code. However, because it is possible that some TNS coding cannot be translated unambiguously before run time, and because P-relative data is not accelerated, the TNS code remains available and will be accessed in those circumstances.

In the case of a nonaccelerated **TNS program** (right sequence in [Figure 5-1](#)), the process, when executing, causes resident millicode to read and interpret TNS instructions from memory. Thus the process appears to be executing TNS instructions.

In both of the latter two cases, the process uses TNS stack conventions (described in [Section 6, TNS Execution Modes](#)) and is therefore categorized as a **TNS process**.

**Figure 5-1. Comparing Native and TNS Processes**

VST260.vsd

# Stacks for Native and TNS Processes

Data stacks are a principal resource of each process. They provide a historical record of procedure calls, as well as the local addressing environment of procedures, including some of the parameters passed between procedures.

In the execution of a native process, two different stacks are used, depending on whether or not privileged mode is in effect. The illustration in [Figure 5-2](#) shows the locations of the two stacks.

The **main stack** is the normal stack area of a nonprivileged native process. This stack area begins at the end of the 4E region, just below virtual address 50000000 (hexadecimal). This address is defined by the operating system. The stack grows from higher addresses to lower addresses. The default maximum size of the main stack varies with the kind of process. The largest main stack currently defined is one full unitary segment (128K bytes).

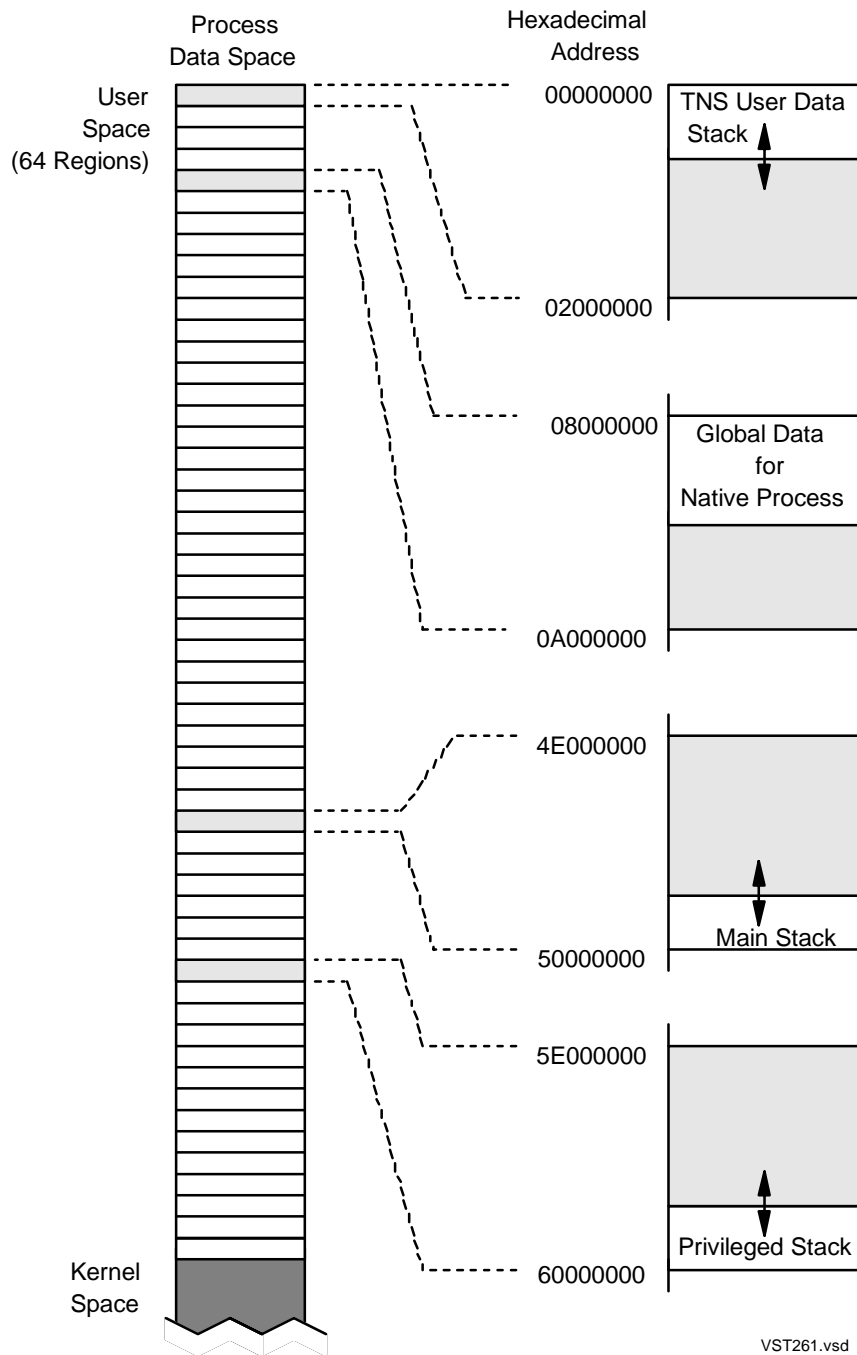
The **privileged stack** is used for native mode procedures running privileged. It is the normal stack area for privileged native processes. Native procedures that are not callable execute on the same stack as their caller. For nonprivileged processes, execution switches to the privileged stack when a native callable procedure is entered. Such a switch requires that the procedure's arguments be moved to the privileged stack from the main stack.

The privileged stack begins at the end of the 5E region, just below virtual address 60000000 (hexadecimal). This address is defined by the operating system. Like the main stack, the privileged stack grows from higher addresses to lower addresses.

When a process is not privileged, the privileged stack is always considered to be empty; any contents will be overwritten. The only exit from the privileged stack to the main stack is through a return from a callable procedure that was called from a nonprivileged procedure. The switch back to the main stack is done as part of the privilege transition return.

For TNS processes, there is an additional stack, termed the **TNS user data stack**. It contains data used by TNS procedures and is located in the first half of the first unitary segment of the first region. This stack is in addition to those described above, because the TNS process switches to native mode, using the main or privileged stack, when calling RISC procedures in the system library.

For native processes, global data is allocated at the beginning of the 08 region. For TNS processes, global data is allocated immediately preceding the TNS user data stack.

**Figure 5-2. A Native Process Uses Two Stacks; TNS Uses Three**

# Code and Data Allocations

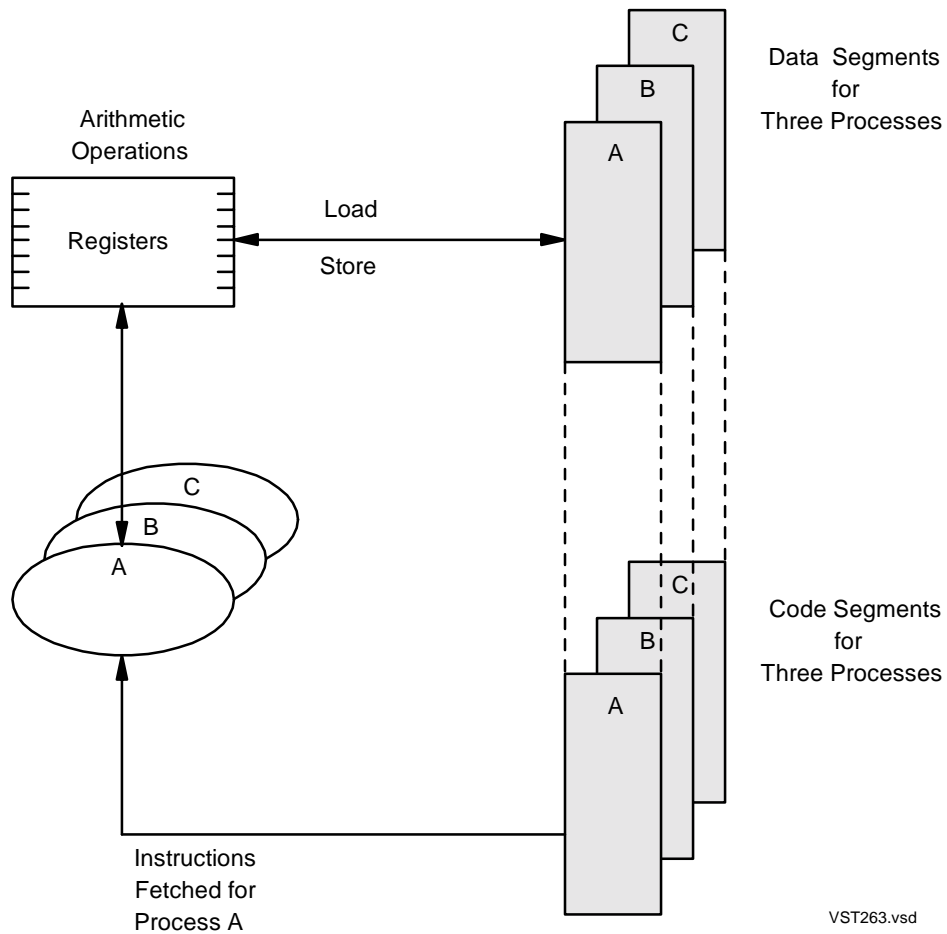
As explained in [Section 4, Memory Addressing and Access](#), each process has its own separate allocation of user space in virtual memory. Within that allocation, segments for code are assigned in the last few regions, and segments for data are assigned in several lower address regions. The code segments contain program instructions and constants in one or more of three possible forms (native code, Accelerator-generated RISC code, or TNS code), and, for accelerated programs, mapping tables. The data segments contain program variables. Each process also has a logical segment called the process file segment (PFS), containing operating system data for the process.

The code segments of a process can be thought of as read-only storage, because no user code instructions can write into them. Accordingly, because code segments cannot be modified, they can be shared by any number of processes. Information in the data segments can be both read and written, using load and store instructions—except for those data segments that are declared to be read-only.

Every process has at least two data segments: the privileged stack segment and the process file segment. If the process is a TNS process or if it is unprivileged, it also has a main stack segment. A TNS process also always has a TNS user data segment (for TNS stack and global data). A native process also has a global data segment.

[Figure 5-3](#) shows a simple example of three processes (A, B, and C), each executing its own set of code segments and accumulating data in its own private data segment (correspondingly labeled).

The process A is executing instructions fetched from code segment A, which is located near the end of the user space in virtual memory. The process is loading data from stack segment A, operating on that data in processor registers, and storing the resulting data back to the data stack segment. Processes B and C, not currently in execution, have their own code segment sets and their own data stack segments.

**Figure 5-3. Code and Data Allocations Are Separate and Treated Differently**

# Sharing of Code and Data Segments

Code segments can be shared. That means that any number of processes can be fetching and executing instructions from the same object code segment. The operating system keeps track of which processes are using which segments. When the final process among several that are using a given code segment ceases to exist, that code segments can (and will) be deallocated by the operating system.

A notable example of sharing code segments is the **system library**, which is a set of code segments that is shared by all processes existing in a given processor. Although there is only one copy of the system library in the processor, each process can access the library as part of its own user space in virtual memory.

The data stack segment (whether main stack, TNS stack, privileged stack, or Debug stack) holds data that must be private to each individual process, as does the global data segment of a native process. Stack data includes the call/return stack chain of activation records that retains the historical state of the process.

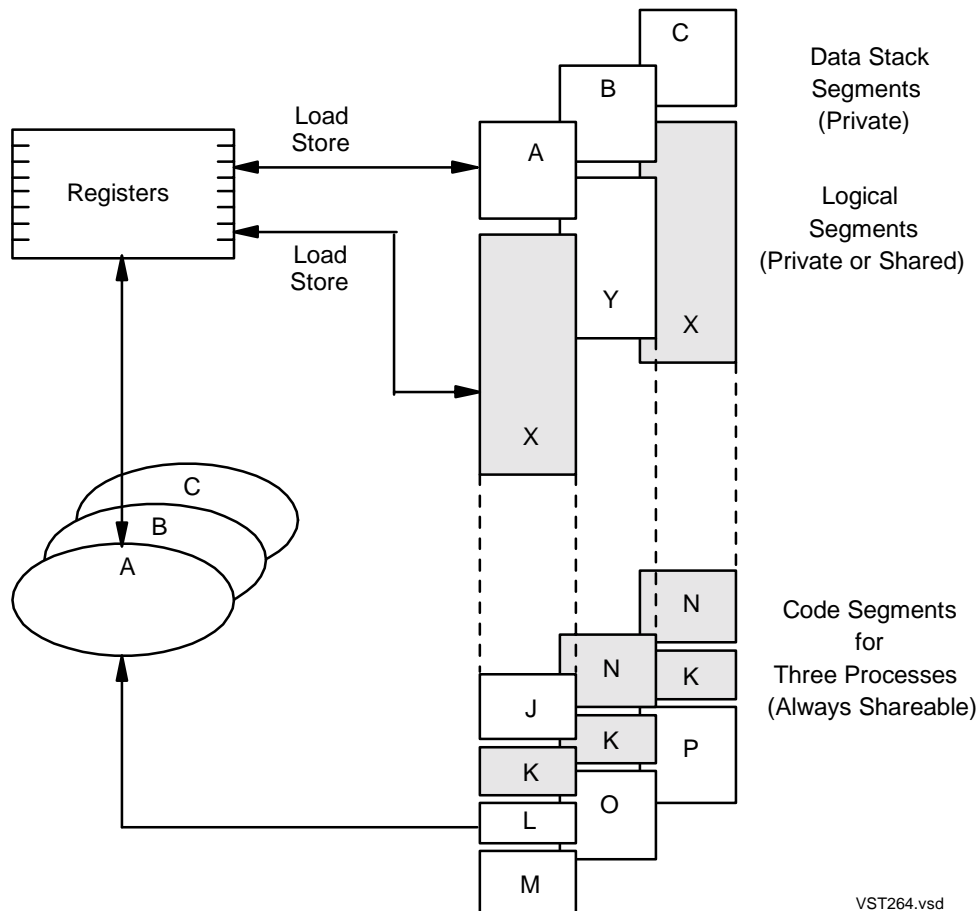
Logical segments that the process establishes with a `SEGMENT_ALLOCATE_` procedure call can be shared—if they are declared to be shareable when they are allocated. (The disk file for a read-only logical segment can also be shared by processes existing in different processors.) In this sense they are like files but, unlike files, logical segments (both selectable and flat) form a part of each process's memory space.

[Figure 5-4](#) shows an example of three processes (A, B, and C), each with its own private data stack segment but (to some extent) sharing code segments and logical segments. The shading indicates those segments that are actually shared. Although these segments appear in the address space of each process sharing them, there is in fact only one copy of each.

Processes A, B, and C (like all processes) share the system library segments, here labeled K in each process's user space. Processes B and C share some other object code segments, labeled N. Processes A and C share a logical segment, labeled X. Process B does not share its logical segment, labeled Y.

When segments are shared by two or more processes, the same pages of data or code are used by all of them, and the addressing tables (SPTs and Vsegs) that designate them are also shared. The operating system keeps track of which processes share each segment (and at which addresses, which can vary from process to process for a shared flat segment).



**Figure 5-4. Code Segments and Some Data Segments Can Be Shared**

# Restricting Mode Transitions

When a process begins execution, the mode of execution depends on the form of the program object code. If the program was compiled as a TNS program, execution begins in TNS mode, wherein a millicode interpreter program executes on behalf of the TNS instructions. If the program was a TNS program that was accelerated, execution begins in TNS mode but quickly switches to accelerated mode, wherein the translated TNS code, in the form of RISC instructions, executes directly instead of the millicode interpreter. If the program was compiled directly to RISC instructions, program execution begins in native mode.

Changes of execution mode are restricted as shown in [Figure 5-5](#). A program executing in TNS mode must first make a transition to accelerated mode if, for example, it calls a native library procedure. Likewise, on return from the native library procedure, execution mode must switch to accelerated mode before returning to the calling point in the TNS process. Such transitions are accomplished within shell procedures. Shell procedures are discussed in detail in [Section 7, Native Execution Mode](#).

In this example there are three programs: A is TNS code, B is TNS code that has been accelerated, and C is native code. Each kind of code executes in its own mode. Accelerated objects usually execute in accelerated mode, but sometimes execute the original TNS code in TNS mode. For system library calls, native code can call only SLr routines, and accelerated code can call procedures in either SL or SLr; however, TNS code can call only SL routines and SL shells that can access SLr.

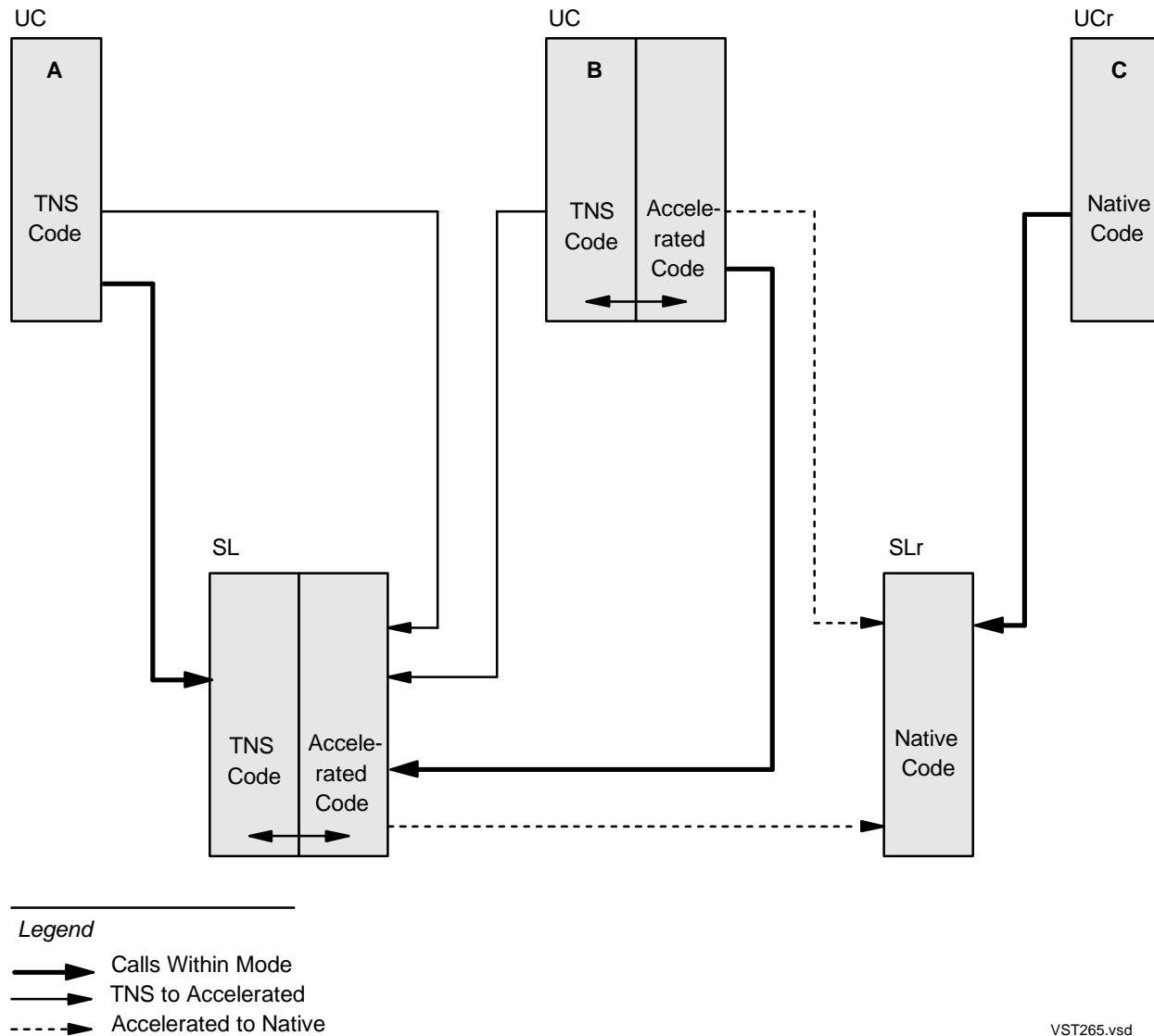
The current execution mode of the current process is maintained as a mode flag in the RP wrap page. Although the RP wrap page is in unprotected virtual memory, protection for the mode flag is unnecessary, because (unlike switching to privileged state), changing execution mode offers no particular privileged advantage.

The state of the mode flag determines which set of stacks is in effect for the current process—either the TNS stacks or the native RISC stacks. The mode also determines how the RISC processor registers are used; different register conventions apply in each mode. TNS and accelerated register usages are similar, using several registers to hold emulated TNS processor state. Native mode usage is quite different.

Execution mode switches between TNSmode and accelerated mode occur at procedure calls and exits; an acceleratedmode to TNS mode switch sometimes occurs within a procedure. Execution mode switches from accelerated to native mode only at procedure calls, and reverts at exit.

The following table summarizes register and stack conventions for the three modes.

Mode Flag	Register State and Usage	Stack Location and Structure
TNS	TNS (nonaccelerated)	TNS
Accelerated	TNS (accelerated)	TNS
Native	Native	Native

**Figure 5-5. Only Two Kinds of Mode Transitions Are Permitted**



---

---

# 6

---

---

# TNS Execution Modes

The following topics describe operations for the TNS compatibility modes.

[Execution Modes for TNS Compatibility](#)

[TNS Addressing Conventions](#)

[The Environment Register](#)

[The Register Stack](#)

[Register Stack Operations](#)

[The Register Stack in Memory](#)

[Basic P Register Operations](#)

[Branching, Direct and Indirect](#)

[Indexed Addressing in a Code Segment](#)

[Direct and Indirect Addressing in the Data Segment](#)

[Byte Addressing in the Data Segment](#)

[Indexing in the Data Segment](#)

[Examples of Indexing in the Data Segment](#)

[SG Addressing Mode](#)

[Basic Characteristics of Procedures](#)

[Procedure Attributes](#)

[Defining the Procedure's Data](#)

[Data Segment Addressing Modes](#)

[Operations at the Procedure's Top-of-Stack](#)

[Overview of Procedure Call and Exit](#)

[Actions of the PCAL Instruction](#)

[Actions of the EXIT Instruction](#)

[A Procedure's Local Variables](#)

[Passing Parameters to a Called Procedure](#)

[Accessing Parameters in the Called Procedure](#)

[Saving the Stack Frame on a Call](#)

[Restoring a Stack Frame on Return From a Call](#)

[Multiple Markers for Nested Calls](#)

[Returning a Value to the Caller](#)

[Retrieving a Returned Value](#)

[Subprocedure Calls](#)

[Calling External Procedures](#)

[Example of an External Procedure Call](#)

[Resolving Virtual Addresses for External Calls](#)

[An Accelerated Program File in Virtual Memory](#)

[Execution Mode Switches](#)

[Procedure Return in Accelerated Code](#)

[Mapping Return Addresses and Debug Points](#)

[Gateway Tables](#)

[Far Jump Tables](#)

[Maintaining TNS State Values](#)

[Invoking Privilege for CALLABLE Procedures](#)

# Execution Modes for TNS Compatibility

This topic describes the basic differences between the two execution modes that provide TNS compatibility.

**TNS mode** means that the interpreter millicode (in resident memory) executes on behalf of the process, interpreting TNS instructions from the object code in memory during run time. The millicode updates process state information as if the process were executing on a TNS processor. **Accelerated mode** means that the object code of the process, translated and optimized prior to run time, is directly executed by the RISC processor. The distinction is illustrated in [Figure 6-1](#).

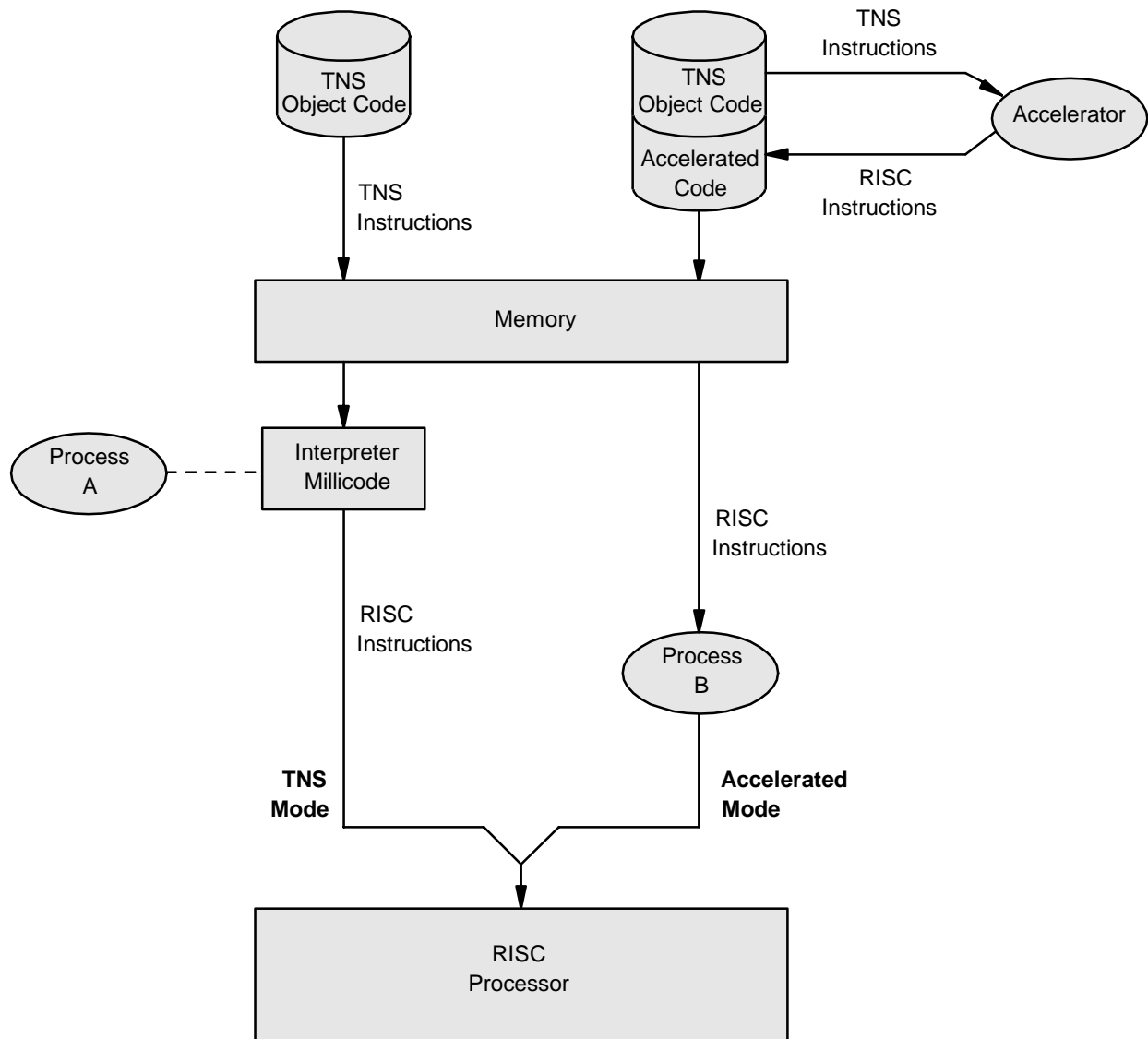
In the operating system environment, processes compete for execution based on operating system rules. All processes that are not currently running preserve flags (in their register-save area) that state whether that process was using TNS or accelerated mode. When a process is dispatched into execution, the instruction processor reads those flags and accordingly fetches instructions from either the millicode (TNS mode) or the program file (accelerated mode). In either case, the RISC processor “sees” only a stream of RISC instructions; there is no switching of hardware or data paths.

In TNS mode, the interpreter millicode reads instructions from a TNS code segment in memory as if they were data, analyzes those instructions, then executes a stream of RISC instructions that perform the appropriate operations and update the saved copies of TNS registers. Thus the state of the process can be made available in the form of emulated TNS registers, register stack, and data stack.

In accelerated mode, the TNS object code has been translated from TNS instructions to RISC instructions, and both forms of the object code exist in the object code file. This work is performed prior to run time by the Accelerator. “Acceleration” of object code is accomplished in several ways. For example, because the translation work is performed “offline,” the Accelerator is allowed to look ahead in the instruction stream and avoid generating RISC instructions for side effects (like setting of condition codes) that will not be used. Also, translation needs to be performed only once for any section of code, whereas the millicode must reinterpret the instructions on each pass through that section.

In accelerated mode, the TNS state is not always readily available. That is because sequences of TNS instructions are translated into equivalent but optimized sequences of RISC instructions that compute the same results directly in RISC registers and memory. The TNS instructions are not translated one-by-one into identical intermediate values. When necessary, however, the corresponding state of TNS registers can be derived at certain well-defined points. For some complex TNS instructions, the accelerated code directly invokes millicode procedures.

In the example shown in [Figure 6-1](#), process A uses TNS mode. The TNS instructions from its object code file are read and interpreted by the millicode. The millicode executes on behalf of the process and maintains process state. Process B uses accelerated mode. Prior to run time, the TNS instructions in its code file were translated and optimized into an accelerated code file. RISC instructions from the accelerated code execute directly.

**Figure 6-1. Two Execution Modes Provide TNS Compatibility**

VST266.vsd

# TNS Addressing Conventions

When TNS mode or accelerated mode is in effect in the processor, both code and data are located in different areas of virtual memory than when native mode is in effect (see [Section 4, Memory Addressing and Access](#)). In addition, these modes use addressing conventions that are specific to the TNS architecture. This and the remaining topics in this section describe these conventions.

The user data segment consists of up to 65,536 16-bit words. Addresses in the data segment start at G[0] (global data, word 0) and progress consecutively through G[65535]. This is shown in [Figure 6-2](#). The TNS user data stack, which dynamically varies in size, is limited to the first half of the data segment, G[0:32767].

The code space of a given process consists of a user code space (UC) and optional user library space (UL). In addition, although not shown, all processes share system library (SL) and system code (SC) spaces. This example shows four user code segments allocated: UC.0 through UC.3.

Each space is a single program file that can contain multiple unitary segments: up to 32 segments of TNS code in each code space. (Also possibly present, but not shown, is the accelerated RISC code if the code was accelerated.) To name the individual segments within these two spaces, a space ID (space identifier) index is used. A space ID index is a number in the range %0 through %37. Examples, as written: UC.0, UL.17, and SL.6.

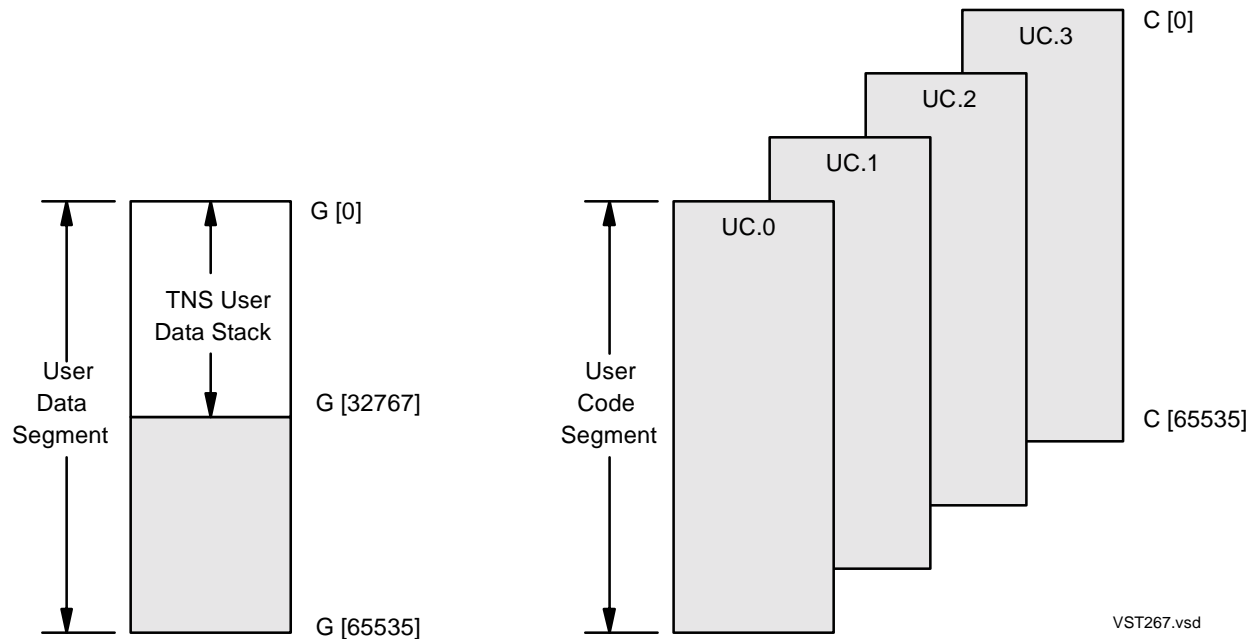
External procedure calls are used to call procedures from one segment to another in all segments of the user code and library spaces, as well as to call procedures in the system library.

Information in a TNS code segment consists of executable code and read-only data. Although it is possible to address the code segments (using extended addressing or instructions such as MOVW, MOVB, COMB, COMW, SBU, SBW, LBP, LWP, and LWUC), only read access is permitted; a write access attempt results in an address trap.

Each TNS code segment occupies up to 65,536 16-bit words in one unitary segment. Words in a code segment are numbered consecutively from C[0] (code, word 0) through C[65535]. This forms the basis for logical addressing within the code segment, as illustrated in [Figure 6-2](#).

Each TNS code segment occupies up to 65,536 16-bit words in one unitary segment. Words in a code segment are numbered consecutively from C[0] (code, word 0) through C[65535]. This forms the basis for logical addressing within the code segment, as illustrated in [Figure 6-2](#).



**Figure 6-2. TNS and Accelerated Modes Use TNS Addressing Conventions**

# The Environment Register

The 16-bit ENV (Environment) register shows the state of the currently executing procedure. That is, the register indicates the code space in which the procedure is currently executing, the data segment that it is using, whether or not privileged mode is in use, whether or not arithmetic traps are enabled, and specific results of the most recent instruction (overflow, carry, and condition code). The format of the ENV register is shown in [Figure 6-3](#). The register pointer (RP) is separately described in the succeeding two topics.

The individual fields of the ENV register are frequently referred to and updated by the operating system and millicode. Most fields are saved (along with the contents of the P and L registers) as part of the executing state of the procedure when it calls some other procedure. (The condition code and register stack pointer fields do not need to be saved, and instead bits 11 through 15 are used to save the space identification of the calling procedure.) Most of the ENV register is restored to its previous state when the called procedure finishes. (All fields in ENV are saved and restored for an interrupt.)

The **library space bit**, or LS bit, (ENV.<4>) works with the CS bit (ENV.<7>) to define the **current code space**. When this bit is equal to 1, it indicates that the current procedure is located in one of the library code spaces (user library or system library) rather than in one of the standard code spaces (user code or system code). The CS bit determines which of the two library spaces is indicated. In the case of “system” indication by CS, the current procedure is in the system library space; in the case of “user” indication by CS, the procedure is in the user library space.

The **privileged mode bit**, or PRIV bit, (ENV.<5>), when equal to 1, indicates that the processor is currently executing in **privileged mode** and is permitted to perform privileged operations. Normally, only the operating system executes in privileged mode. Nonprivileged programs can perform privileged operations only indirectly, by calling procedures designated as callable. When a nonprivileged procedure calls a callable procedure, its nonprivileged state is restored on return.

The **data space bit**, or DS bit, (ENV.<6>) defines the current data segment. This specifies which data area is to be accessed when a data reference is made to relative segment 0. DS, when equal to 0, specifies the user data segment; when equal to 1, it specifies the system data segment. DS equals 1 only in the interrupt environment.

The **code space bit**, or CS bit, (ENV.<7>), together with the LS bit (ENV.<4>), defines the current code space. CS, when equal to 0, specifies the user code space (or the user library space if LS is equal to 1); CS equal to 1 specifies the system code space (or the system library space if LS is equal to 1).

The **trap enable bit**, or T bit, (ENV.<8>) controls the overflow trap, enabling it when T = 1 or disabling it when T = 0. The T bit can be set to either state by the SETE instruction. When the trap is enabled and an overflow occurs (see “Overflow Bit” later in this topic), control is transferred to the arithmetic overflow interrupt handler. When the trap is disabled, the program may choose to ignore the overflow bit or test it for specific handling.

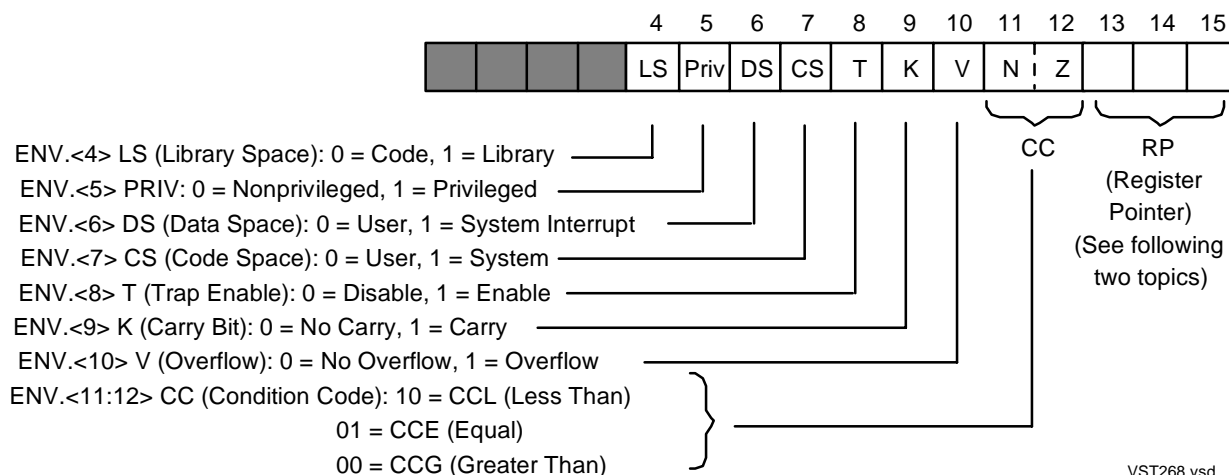
The **carry bit**, or K bit, (ENV.<9>), when equal to 1, generally indicates that a carry out of the high-order bit position occurred when an add or subtract instruction was executed on a 16-, 32-, or 64-bit operand. It reflects the result of the last such instruction executed. Other uses of the K bit exist, such as to specify the result of executing a scan instruction (SBW or SBU).

The **overflow bit**, or V bit, (ENV.<10>), if equal to 1, indicates that an overflow condition occurred or a divide by zero was attempted. Overflow is generally associated with arithmetic operations on 16-, 32-, and 64-bit operands. Overflow also occurs in an LDIV instruction if the quotient cannot be represented in 16 bits, or in floating-point arithmetic if the exponent is too large or too small. The V bit is not “sticky;” it reflects only the most recent instruction that can set or clear it.

The **condition code bits**, a two-bit field, (ENV.<11:12>) forms the condition code. N means negative or numeric; Z means zero or alphabetic. The condition code typically reflects the outcome of a computation, comparison, bus transfer, or input-output operation. Some other common uses are to reflect the outcome of calls to system procedures and, by “load” instructions, to identify the characteristics of the word, doubleword, or byte loaded onto the register stack. Various other uses are specified in the instruction set definition. The condition code has three states:

CCL	= less than,	ENV.<11:12> = 10 (N = 1, Z = 0)
CCE	= equal to,	ENV.<11:12> = 01 (N = 0, Z = 1)
CCG	= greater than,	ENV.<11:12> = 00 (N = 0, Z = 0)

**Figure 6-3. The Environment Register Maintains Procedure State**

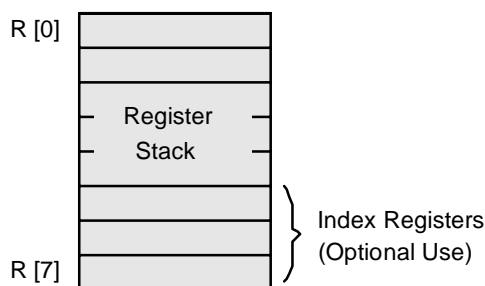


# The Register Stack

To maintain compatibility with other earlier systems, NonStop S-series processors provide a register stack that is used mostly in nonaccelerated (TNS) execution mode.

The register stack consists of eight 16-bit registers, designated R[0] (register stack, element 0) through R[7]. See [Figure 6-4](#). The register stack is where arithmetic computations are performed and where most comparisons are made. Typically, operands are loaded onto the stack, arithmetic operations are performed, the operands are deleted, and a result is left on the stack. Three registers, R[5:7], also double as index registers.

**Figure 6-4. The Register Stack Accumulates Arithmetic Results**

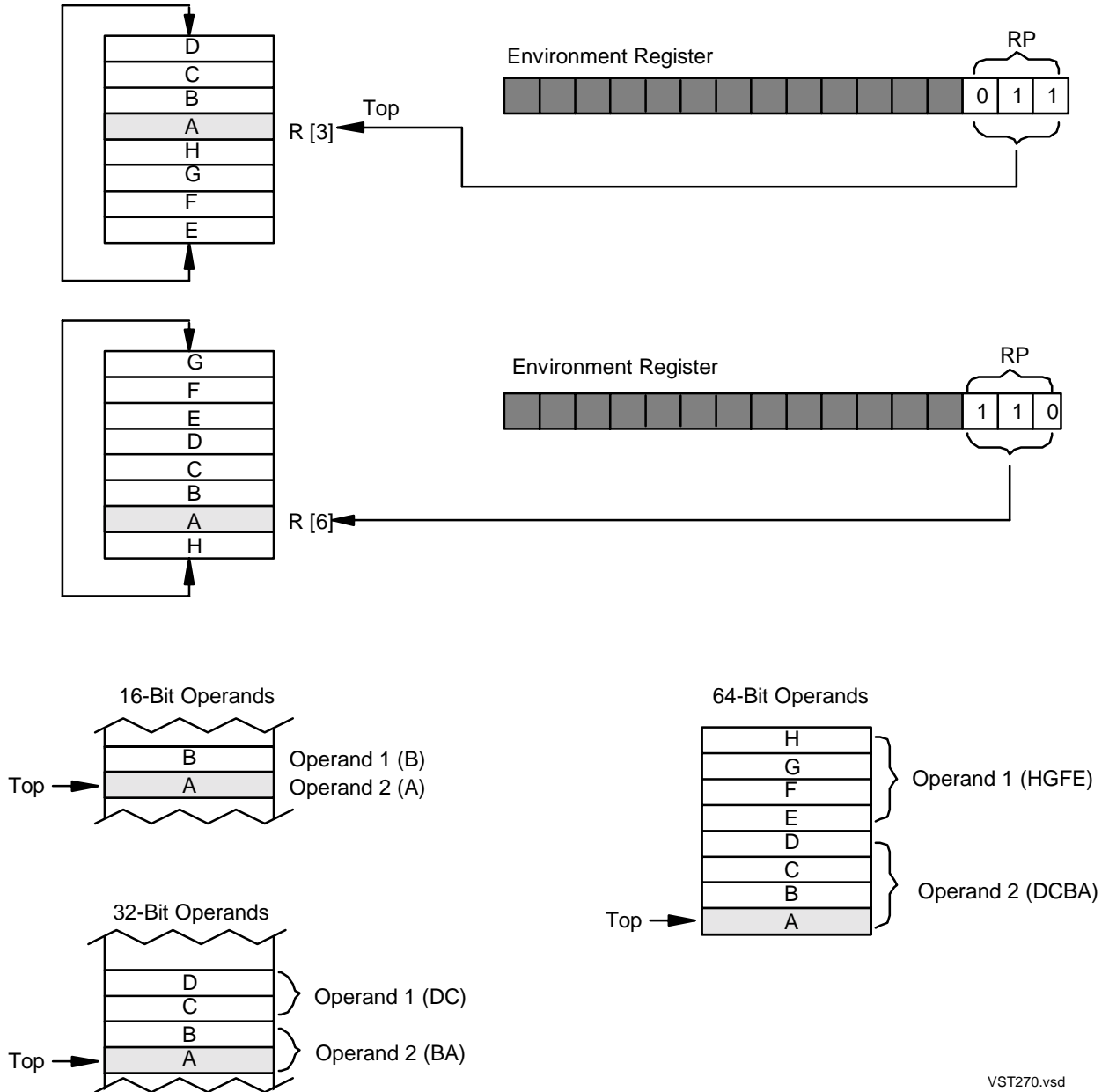


VST269.vsd

Usually, elements in the register stack are addressed implicitly. That is, an instruction operates on the top element (or elements) without specifying an absolute register number. The current top element of the register stack is defined by the **register stack pointer**, RP, shown in the upper part of [Figure 6-5](#). RP, which is a three-bit field in the Environment register, contains the register number, 0 through 7, of the top element. The RP setting is incremented when operands are loaded onto the register stack and decremented when arithmetic is performed or results are stored. The empty (or full) state of the register stack is defined as RP = 7. There is no protection against rolling over from 7 to 0 or from 0 to 7.

The elements in the register stack are named as to their location relative to the current top element. As shown in [Figure 6-5](#), the top element is designated “A”, the second is “B”, and so on through “H”. These names have no fixed relationship to the register numbers, and the naming sequence wraps around the end of the stack. In cases of doubleword and quadrupleword operands, the low-order word is in A.

In the first example in [Figure 6-5](#), because the register pointer in the Environment register contains the value 3, the top of the register stack is R[3]. That register is named A, and the next lower-numbered registers are named B, C, and so on, wrapping around the end of the stack to H. In the second example, RP is 6, so A is R[6]. Three examples of operand pairs on the register stack are also shown.

**Figure 6-5. The Top-of-Stack Can Occupy Various Positions in the Register Stack**

# Register Stack Operations

A typical operation to add two numbers in the register stack is illustrated in [Figure 6-6](#). The operation proceeds as follows: the operands are first loaded from global data (G) into the register stack using LOAD instructions, then an IADD (integer add) instruction is executed to perform the desired arithmetic, and finally the result is stored back into memory using a STOR instruction. Grouped together to form a program, the operation looks like this:

```
LOAD G + 002      !load data element G[2] onto register stack
LOAD G + 003      !load data element G[3] onto register stack
IADD              !integer add
STOR G + 004      !store result from register stack into G[4]
```

Before the operations begin, as shown in the top diagram of the figure, the register stack is assumed to be empty; RP indicates R[7].

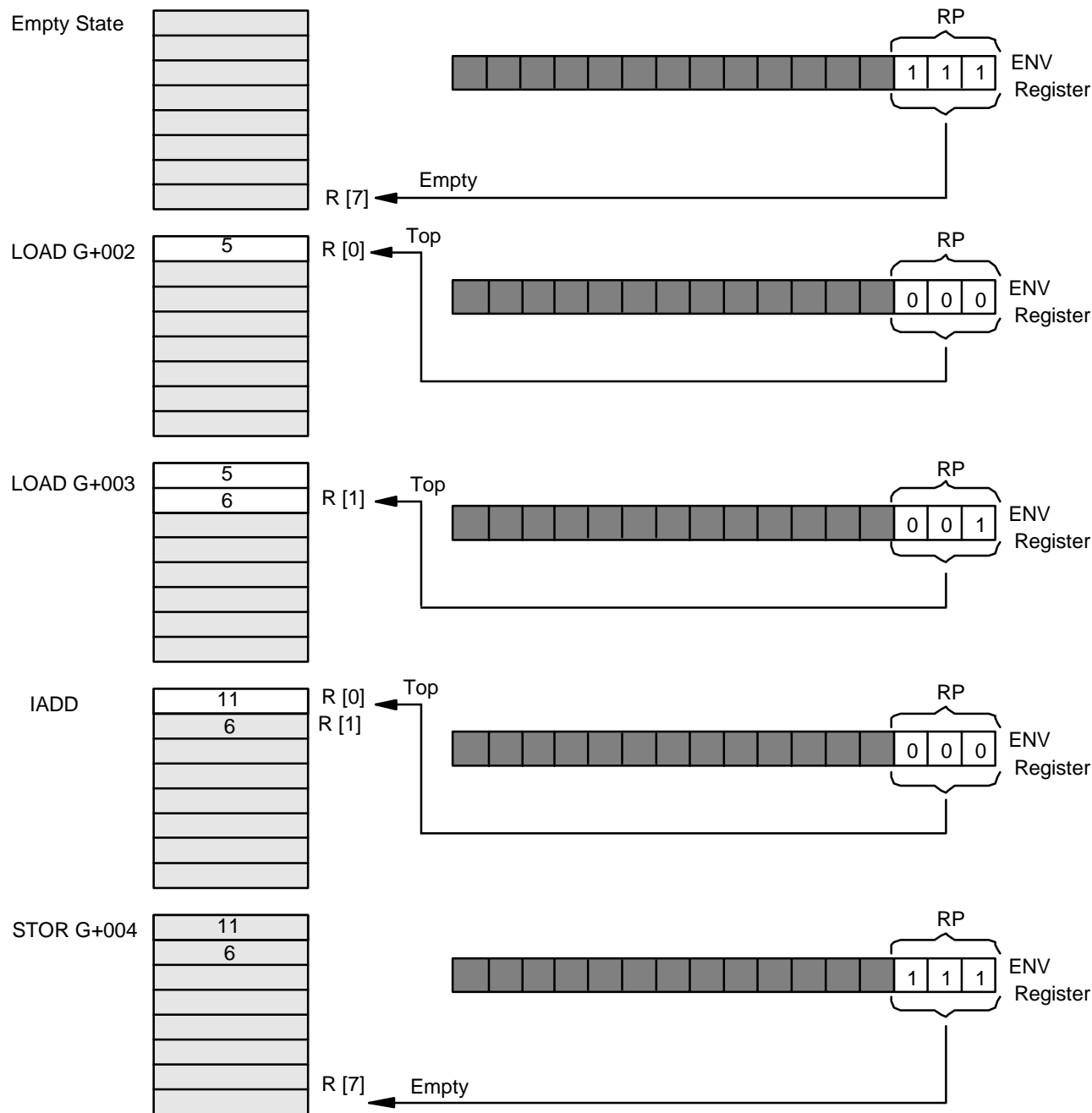
In the second diagram, the value 5 is loaded from the global data area of the user data segment (location G+002). The data item is loaded into the first available stack register, R[0], and RP is incremented to point at this new top-of-stack register.

In the third diagram, the value 6 is similarly loaded from global data. This item is loaded into the next available stack register, R[1], and RP is again incremented to point at this register.

The fourth diagram reflects the condition of the register stack after an integer add (IADD) instruction has taken place. Both operands have been logically deleted by decrementing RP back by two elements (to R[7] again), and the result of the addition has been loaded into the first available stack register. In this case, the first available register is R[0], so the arithmetic result overlays the value that was formerly in R[0], which was the first operand, the value 5. The second operand, the value 6, actually remains in R[1], but because RP now indicates that the top of the stack is R[0], anything in R[1] is logically an undefined value.

The fifth diagram reflects the condition of the register stack after the STOR G+004 instruction has taken place. The arithmetic result of the addition has been stored into memory in global data location G+004, and the value has been deleted from the register stack by decrementing the RP pointer back to 7. That again indicates the empty state. Registers R[0] and R[1] still contain 11 and 6, but succeeding operations will automatically overlay their contents.

**Figure 6-6. An Example of Register Stack Operations**



VST271.vsd

# The Register Stack in Memory

As implemented in the NonStop S-series processor, the register stack (in TNS mode) consists of eight dedicated locations in virtual memory, rather than eight physical hardware registers. The location for this simulated register stack is found in the last segment of every user data space. (Region 63 is a shared region among all processes.) Therefore, in those cases where it is necessary to preserve the contents of the register stack when one process is temporarily suspended to dispatch another, the contents are saved in the process control block (PCB) of the suspended process.

[Figure 6-7](#) shows the structural layout of the register stack in virtual memory. The register stack is located in a single page, called the **RP wrap page**. The current execution mode flag (TNS, accelerated, or native), is also kept in the same page as the register stack, but the primary use of the page is to contain the register stack. A bit in the execution mode flag indicates whether the RP wrap registers are valid (in use). They are valid in TNS mode and in accelerated mode during mode transitions.

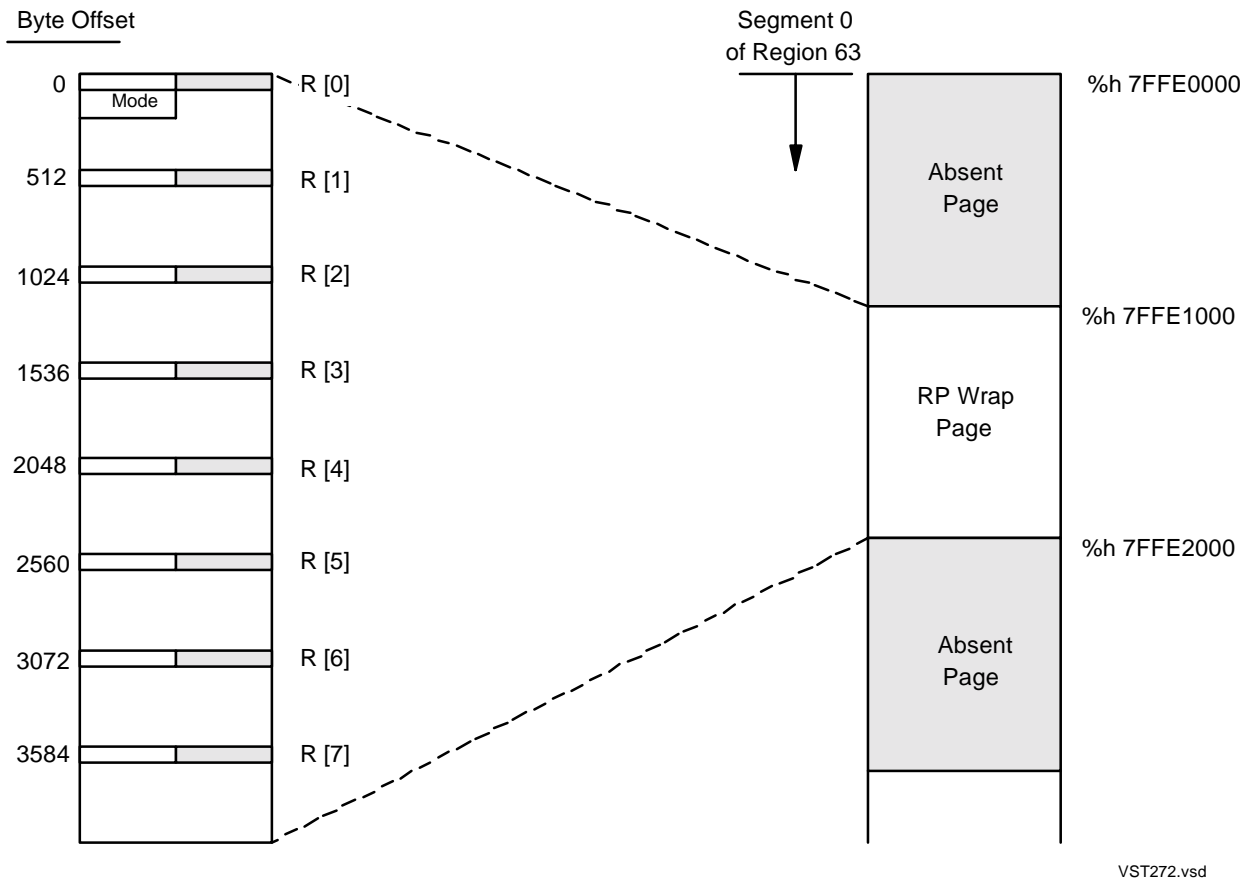
In [Figure 6-7](#), the RP wrap page is the second page in the segment. Within that page, the eight RP registers are kept at byte offset 0 and at succeeding 512-byte intervals from there on. The register data is kept in the lower half of the 32-bit RISC word; the upper half is undefined.

The pages immediately preceding and following this page are absent pages; that is, every entry in the segment page tables for these segments has the Valid indicator flag permanently set off (meaning absent). This is done because it is possible for **RP overflow** or **RP underflow** to occur, wherein an attempt is made to set RP to a value of 8 or greater or to a value of -1 or less. In those cases, the reference will fall in one of the absent pages. If a register access is attempted to a register out of range, an exception occurs; the exception handler recognizes the address as being on one of those two special pages, adjusts the register pointer by 8 in the appropriate direction, and continues. (No exception occurs as the register stack switches to and from the empty state, RP = -1, as long as no instruction attempts to access the registers.)

RP overflow or underflow is possible in this processor because the RP index is kept as an address in a 32-bit RISC register. That is unlike the RP pointer in the TNS architecture of earlier systems, in which RP is merely a 3-bit field in the Environment register and simply rolls over from 0 to 7 or from 7 to 0. In NonStop processors, such rollovers need to be detected by an exception so that the RP address can be adjusted.

In accelerated mode, registers are kept in the RP wrap page only during transitions to TNS mode. Instead, the TNS register values currently significant are retained in RISC registers.



**Figure 6-7. The TNS Register Stack Is Implemented in the RP Wrap Page**

## Basic P Register Operations

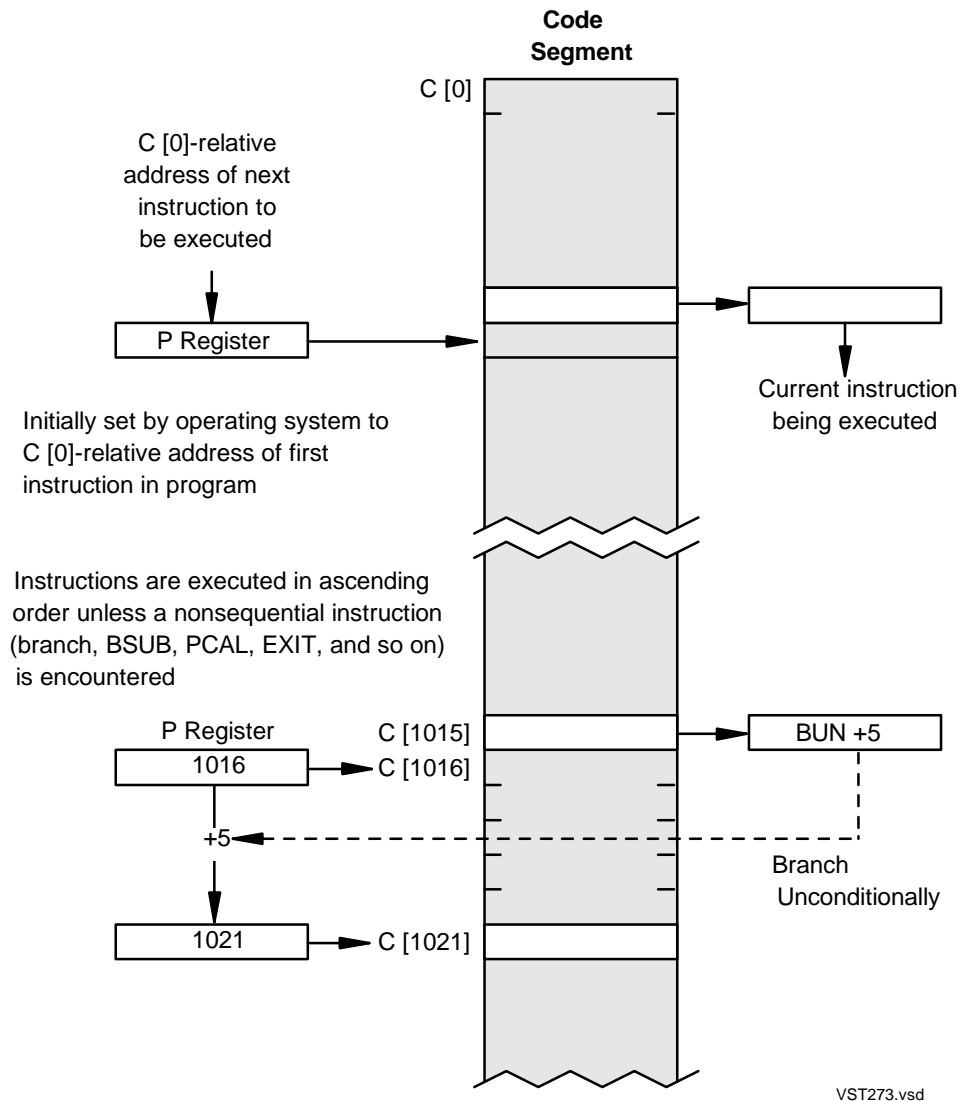
The **P register** (Program register) is the program counter of the TNS environment. It contains the 16-bit C[0]-relative address of the next instruction to be executed. Conventionally, the contents of the P register are incremented by 1 at the beginning of instruction execution so that, ordinarily, instructions are fetched (and executed) from ascending memory locations. This is shown in the upper part of [Figure 6-8](#).

When a program branch is taken or a procedure or subprocedure is called, the C[0]-relative address of the next instruction to be executed is placed in the P register. Execution then resumes sequentially from this new point. A simple branching example is shown in the lower part of the figure. In this example, the P register at C[1015] causes an unconditional branch (BUN) instruction to be fetched, and advances to C[1016]. The execution of the BUN instruction causes a value of 5 to be added to the current P register value, resulting in a new P value and a jump to C[1021].

Both of these cases assume that TNS instructions are executing in a TNS environment. Instead, only RISC instructions actually execute, and strictly in the RISC environment. All TNS instructions must either be translated to RISC instructions prior to run time or be interpreted during run time. Translation to RISC instructions (if done at all) is performed by the Accelerator.

The RISC processor has its own PC register that, generally speaking, increments several times for each single increment of the TNS P register. Thus the correspondence of TNS P to RISC PC is often difficult to ascertain. To further complicate matters in the case of accelerated code, optimization of a group of TNS instructions frequently results in a corresponding group of RISC instructions. This action blurs any instruction-for-instruction association. Thus, conversion from TNS P to RISC PC (a topic considered later in this section) can be done only at certain definable points, and the inverse conversion is precise only at those points.

TNS P is exact in nonaccelerated mode, except for interrupt handlers and memory access debug points. RISC PC can always be ignored in TNS mode.

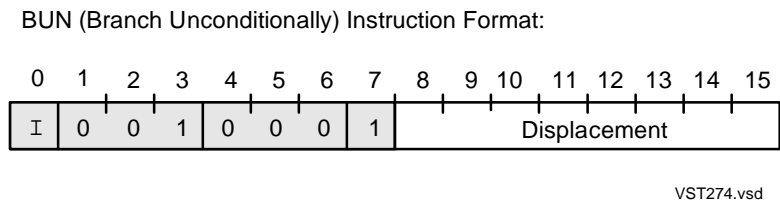
**Figure 6-8. In Concept, the P Register Controls Execution Flow**

# Branching, Direct and Indirect

Addresses for branching (and for constants) in a code segment are calculated relative to the current setting of the P register. This is referred to as **P-relative addressing**.

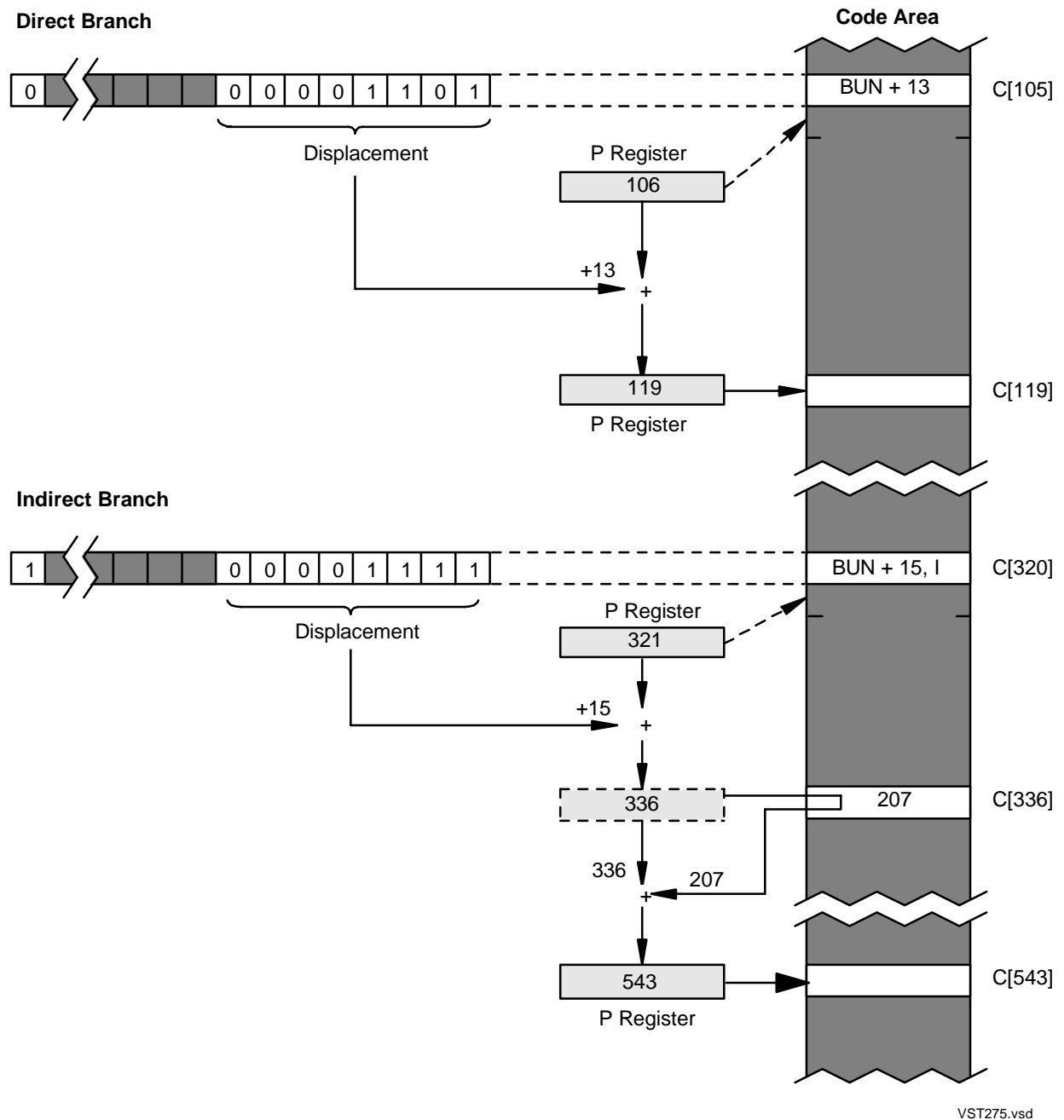
Instructions that reference a code segment have an eight-bit field (seven magnitude bits plus a sign) for specifying a relative displacement from the current P register setting. The range of the displacement is therefore  $-128:+127$  words. An example, the BUN instruction, is shown in [Figure 6-9](#).

**Figure 6-9. The BUN Instruction Is Typical of Branch Instructions**



A branch can be either direct or indirect (an example of each is shown in [Figure 6-10](#)). In computing a branch target address, compilers first add the displacement to the current P register setting. This value is the **direct branch address**. If the referenced location is within the range of the displacement (that is, within the range P  $[-128:+127]$ ), then **direct addressing** is indicated and the direct branch address is used as the branch address. Direct addressing is specified when the **indirect bit** (bit 0) of the instruction is equal to 0.

If the referenced location is beyond the range of the displacement, the compiler generates an **indirect branch** and the referenced location is a relative displacement from the direct branch address. That is, the contents of the word at the direct branch address (containing a displacement from itself) are added to the direct branch address. The result is the target C[0]-relative branch address. Because a 16-bit word is available to specify a displacement (not just seven bits, as in direct addressing), the jump range is extended to the entire TNS code segment. **Indirect addressing** is specified when the indirect bit of the instruction is equal to 1.

**Figure 6-10. Two Examples of Branching**

# Indexed Addressing in a Code Segment

In addition to direct and indirect addressing, an offset value in one of three **index registers** can be added to the address of the direct or indirect location before the final address is calculated. This permits a code segment location to be referenced as an offset from a base location; this is called **indexing**.

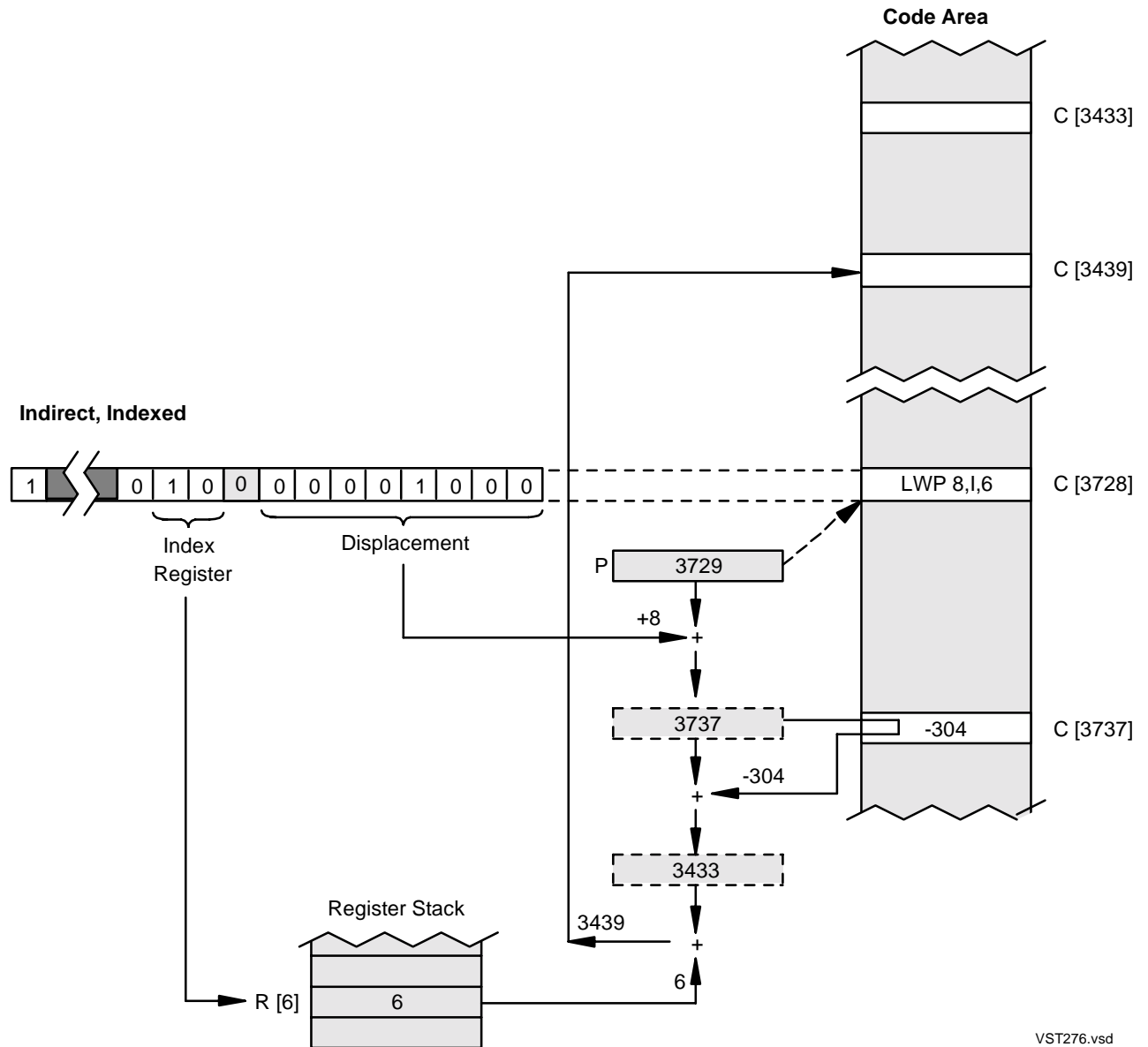
Indexing in a code segment is not used for branching but rather for accessing read-only arrays compiled into the code segment. [Figure 6-11](#) is an example that illustrates the action of an LWP (Load Word from Program) instruction. It shows a combination of indirect and indexed addressing.

At the start of this example, the current P register value is 3729. The displacement value in the instruction is 8. Adding these together produces a direct reference address of 3737. Because indirect addressing has been specified (note the 1 in the most significant bit of the instruction), the content of location C[3737] is taken as a new, self-relative displacement value. That value is negative in this case (–304), so the addition of this value to its own address (3737) produces a new address of 3433. This is a lower address than the current P register setting, and it indicates the start of the array (element 0).

To select one element of the array, the content of an index register is needed. In this case, the instruction coding specifies the value 2 in its index register field. This means that the second of the three index registers contains the index value. As described earlier, the last three registers of the register stack (R[5], R[6], and R[7]) double as index registers. Thus the second of these three is R[6], and the content of that register is taken as the index value.

In this case the index value is 6, so 6 is added to the base address of the array (3433) to finally address location C[3439], the address that contains the desired element of the array. The content of this location is the information that the LWP instruction loads into one of the other locations of the register stack (whichever register is becoming the top of the stack).

Addressing of byte elements (with indexing) is also permitted in the code segment, though restricted to only half of the segment. The reason for this restriction is that one bit of the address is used as a byte specifier, thus cutting the address range by half. For the direct, indexed case, the half segment is the same half in which the current P register setting is located. For the indirect, indexed case, the half segment is the one containing the indirect cell. Byte addressing in the code segment is accomplished by the LBP (Load Byte from Program) instruction.

**Figure 6-11. An Example of Code Segment Indexing**

# Direct and Indirect Addressing in the Data Segment

A TNS process has only one segment for its global data, called the **user data segment**. It contains the process's global variables and data stack.

Various addressing modes are provided for addressing locations relative to the current stack frame; however, those modes will be covered in later topics, when procedure calls are described. For now, assume there is only one mode, the simplest one—the **G-relative mode**. This is the “global” mode, in which all addresses are relative to G[0], the starting address of the segment.

In any instruction that refers to the data segment, there is a mode specifier and a displacement field, occupying bits 7 through 15 of the instruction word. This is shown in [Figure 6-12](#). In this case, a 0 in bit 7 specifies the G-relative addressing mode, and bits 8 through 15 specify the displacement from G[0]. That field size limits the displacement range to 256 words; that is, G[0] through G[255].

If the indirection bit (bit 0) of the memory reference instruction is equal to 0, then direct addressing is specified. With direct addressing, the address of an operand referenced by an instruction is directly specified in the address field of the memory reference instruction. Only one memory (or data cache) reference is needed to access the referenced memory location. If, in the example shown, the indirection bit were 0, location G[11] would be the referenced location.

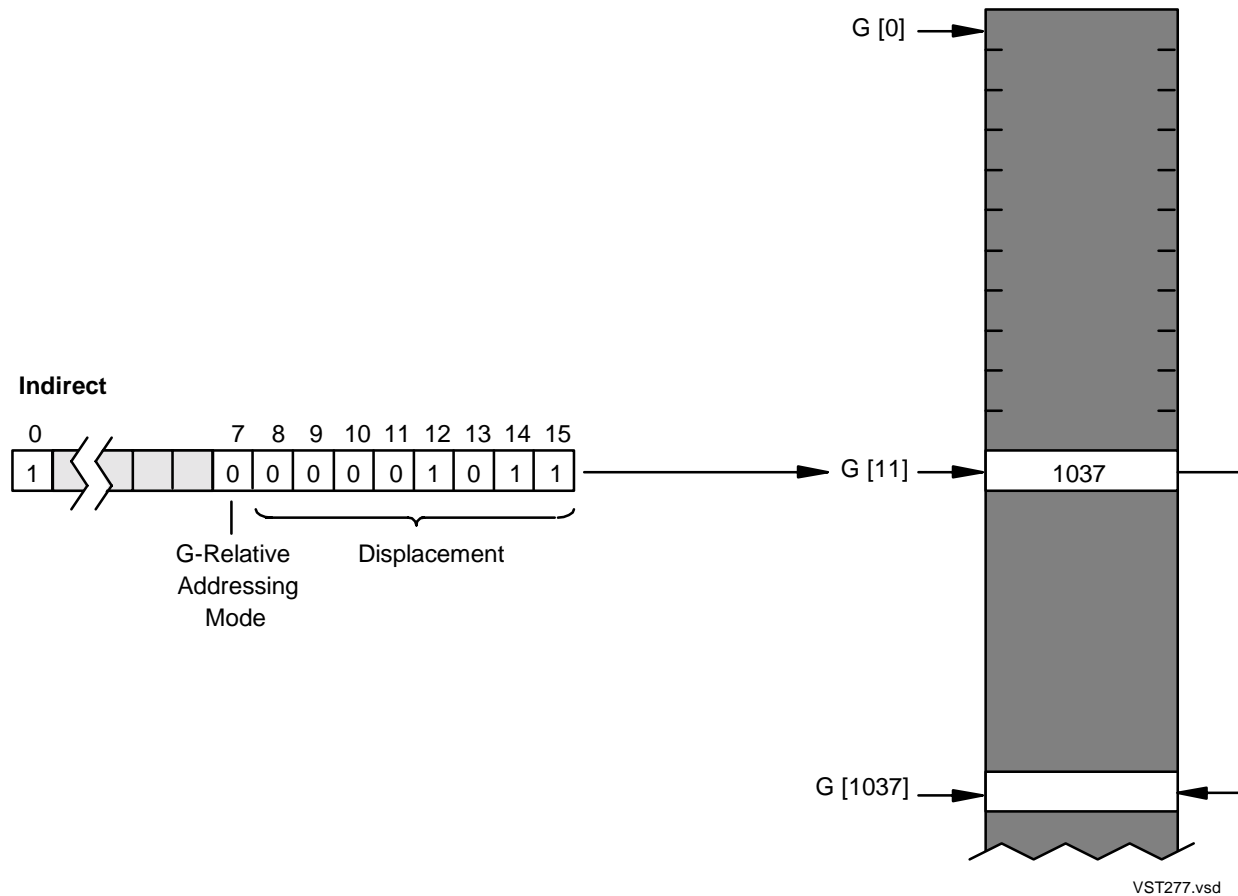
However, if the indirection bit of the memory reference instruction is equal to 1, then indirect addressing is specified. With indirect addressing, the address of the referenced location, relative to G[0], is contained in a location that can be addressed directly. The contents of the direct location are used as an **address pointer**. (This differs from the case of indirect addressing in a code segment, where the direct location contains a self-relative offset, not an address pointer.)

In the example shown, the direct location is G[11]. That location contains the value 1037, so G[1037] is the location accessed by the instruction.

Two memory references are needed to access the referenced location: the first to fetch the address and the second to access the operand. Because the address pointer is 16 bits (rather than just 8 as in the displacement field of the instruction), the range of indirect addressing is G[0:65535]. That means that indirect addressing can access any location in the data segment.

If the instruction applies to a doubleword operand, the operand consists of two words starting at the referenced location—whether arrived at directly or indirectly. Using ordinary memory reference instructions, quadruplewords cannot be accessed as such in the data segment. A quadrupleword must be accessed as some combination of smaller units, such as two doublewords or four words. (However, stack-operand instructions, such as QLD and QST, can load and store entire quadruplewords.)



**Figure 6-12. An Example of Indirect Addressing in the Data Segment**

## Byte Addressing in the Data Segment

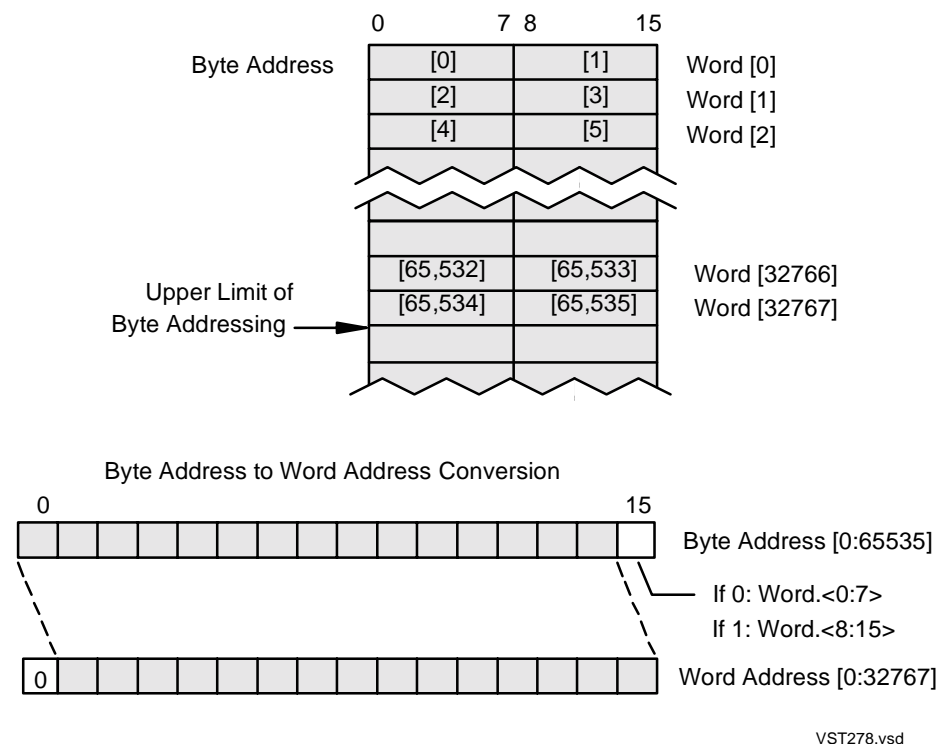
Two bytes can be stored in a 16-bit word. The most significant byte in a word occupies the left half (bits 0 through 7), and the least significant byte occupies the right half (bits 8 through 15).

Memory reference instructions always specify a TNS-word displacement for direct addressing, including those used for byte addressing (LDB, STB, LBP). For direct addressing, the left byte of the word at the displacement location is the addressed byte.

When indirect addressing is used, the word at the displacement location holds a **byte address**. The processor converts this byte address to a word address and bit field in that word, as shown in [Figure 6-13](#). In other words, bit 15 of the byte address is extracted and used to specify left byte (0) or right byte (1); the remaining 15 bits are logically shifted right by one bit to form the word address. In this case, byte-addressable locations in the data segment start at BYTE[0] and extend through BYTE[65535], ending at word location 32767.

Because one bit of a 16-bit byte address must be used to specify left or right byte, 15 bits are left to specify a word. That limits the byte addressing range of a 16-bit byte address to the first half of the data segment.

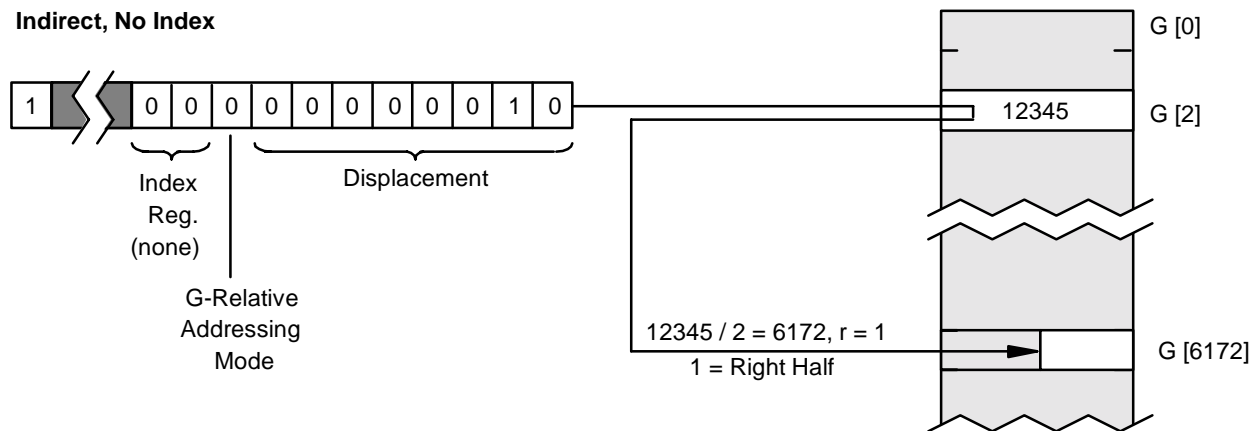
**Figure 6-13. Byte Addressing in the Data Segment Is Restricted to Half Segment**



In the example shown in [Figure 6-14](#), indirect addressing is used to get a byte address. The (word) displacement value is 2, which means that the address is in G[2]. In this case, G[2] contains the value 12345, which is a byte address. To convert that to a word address and byte specifier, the value is divided by 2, giving a word address of 6172 with a remainder of 1. The remainder of 1 specifies the right-hand byte, as indicated in the figure.

For byte addressing in the data segment, the displacement field specifies a word displacement from G[0] to a location in the global area. In the case of direct addressing, the left byte of that location contains the addressed byte. In the case of indirect addressing, the content of the word at the specified displacement location contains a byte address that specifies a particular byte (either left or right) in the first half of the data segment.

**Figure 6-14. Indirect Byte Addressing in the Data Segment Can Specify Odd Bytes**



VST279.vsd

# Indexing in the Data Segment

Indexing is used to reference memory locations relative to a data element in memory. A typical use is the accessing of an element in an array.

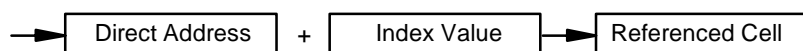
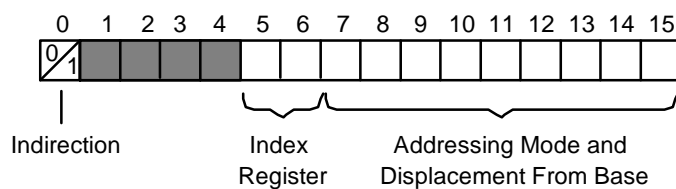
Generally, indexing is done as follows. An initial address is first calculated as described in the preceding two topics (any addressing mode as well as direct, indirect, and byte addressing is permitted). This initial address is then used as a base address for indexing. The indexing value, contained in an index register (referred to as “X”), is added to the initial address to provide the address of the referenced operand. This is shown in the upper part of [Figure 6-15](#).

Any one of three registers in the register stack (R[5:7]) can be used as an index register. The register to be used for indexing is specified in the index register field (bits 5 and 6) in all memory reference instructions. (Note the instruction format in the lower part of the figure.) The index register field corresponds to register stack elements as follows:

Index Register Field	Index Register
0	X = no indexing
1	X = R[5]
2	X = R[6]
3	X = R[7]

An index register can contain values from –32,768 through +32,767 to provide direct word and doubleword addressing of any location in the data area (all addressing is modulo 65,536). Byte addressing, as described in the preceding topic, applies only in the first half of the data segment.

The value in an index register is always treated as an element indexing value. That is, if a byte instruction is being executed, the content of an index register is treated as a byte offset; if a doubleword instruction is being executed, the content is treated as a doubleword offset. The next topic shows examples of word and byte indexing.

**Figure 6-15. Sequence and Encoding for Indexing in the Data Segment****Direct, Indexed****Indirect, Indexed****Instruction Format**

0 = No Indexing

1 = R [5]

2 = R [6]

3 = R [7]

VST280.vsd

# Examples of Indexing in the Data Segment

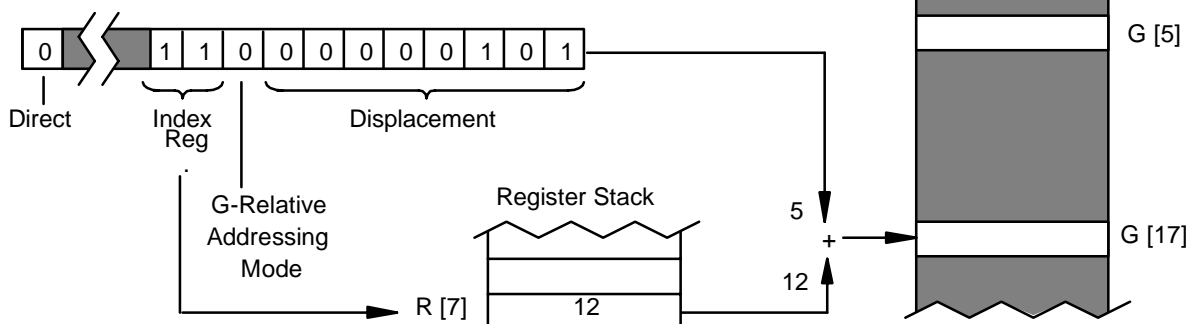
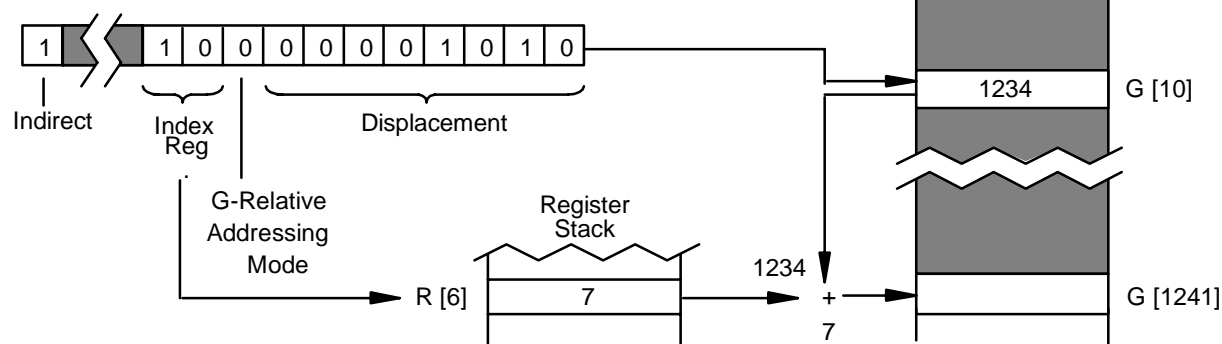
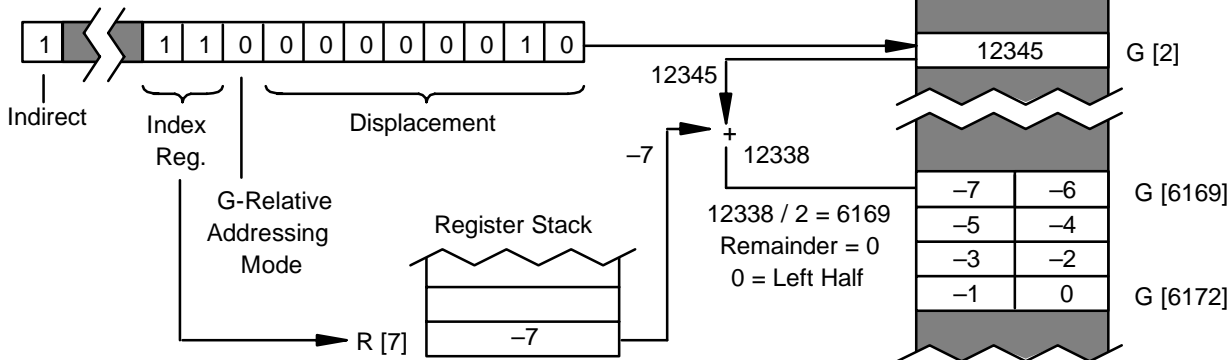
[Figure 6-16](#) illustrates three examples of indexing in the data segment. The first shows indexing with a direct address, the second shows indexing with an indirect address, and the third shows indexing with a byte address—with indirection. For simplicity, the G-relative addressing mode is assumed for all three examples; that is, all offsets for the direct address are relative to the start of the segment, G[0]. Also assumed is that the intent of these instructions is to access an element in a data array.

In the first example, the instruction code specifies direct addressing: bit 0 contains a 0. It also specifies that the third of the three index registers (R[7]) contains the index value: the index register field contains the value 3 (binary 11). The offset is 5 (binary 101). To compute the referenced address, the offset value (5) is taken as the base of the array, at location G[5], and the value in the specified index register (12) is added to that base to arrive at the element address, location G[17].

In the second example, bit 0 of the instruction code specifies indirect addressing (contains 1). The index register field contains the value 2, thus specifying the second of the three index registers, R[6]. The offset is 10 (binary 1010). Computation of the referenced address occurs in two steps. First, the content of the location at the direct address, specified by the offset value (10), is read and taken as the base of the array. As indicated, the base address is 1234. As the second step, the content of the specified index register (7) is added to the base address to arrive at location G[1241].

In the third example, which is assumed to be a byte-referencing instruction, indirect addressing is again specified. The third of the three index registers is specified, and the offset is 2. The index value is a negative number, -7. The first step in computing the referenced byte address is to read the content of the location at the direct address; the direct address is G[2], as specified by the offset, and the content is 12345. This value, 12345, is the base of the array, given as a byte address. Adding the content of the specified index register (-7) to this base gives an element address of 12338. To convert this byte address to a word address plus byte identifier, it is divided by 2, giving a word address with a remainder of 0. The word address, indexed negatively from the base, is location G[6169], and the byte specified is the left half of that word.

In the case of doubleword operands, the indexing value is a doubleword. It is multiplied by 2 to provide a word index. This value is added to the appropriate base address (also a word address): either to the initial address for direct addressing, or to the indirect address pointer for indirect addressing.

**Figure 6-16. Three Examples of Indexing in the Data Segment****Word:****Direct, Indexed****Indirect, Indexed****Byte:****Indirect, Indexed**

VST281.vsd

# SG Addressing Mode

For memory-reference instructions, an additional addressing mode is provided, called the **SG-relative mode**. This mode is provided to permit access, from the user environment, to system global data in the system data segment. This mode directly addresses the first 64 locations of the system data segment (SG[0:63]), and indirectly can address the entire 64K-word segment. Refer to [Figure 6-17](#).

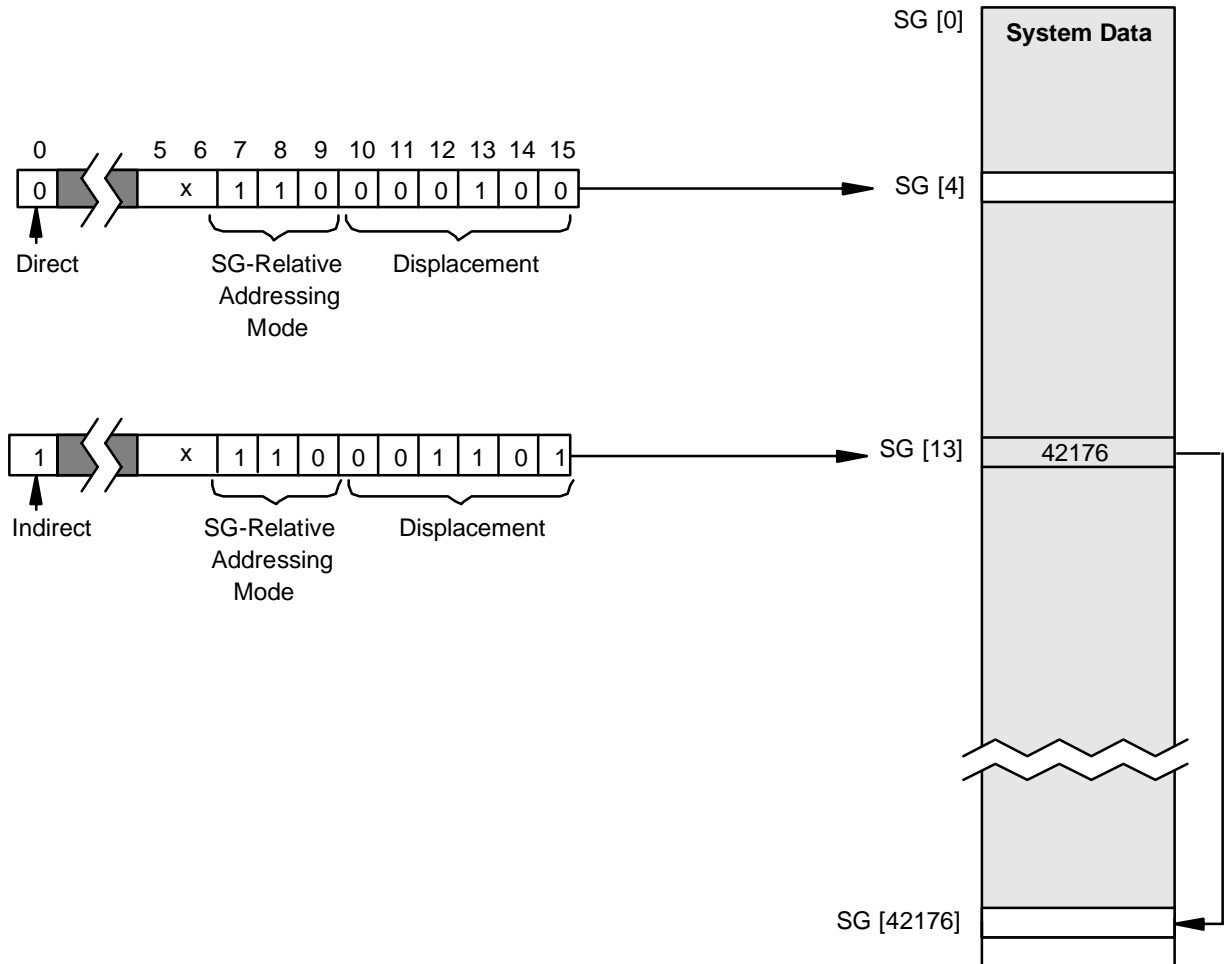
The SG-relative mode is indicated when bits 7 through 9 of a memory reference instruction are equal to 110 (binary). Bits 10 through 15 are a positive word displacement from SG[0]. The specification of SG-relative addressing by the instruction overrides the DS bit setting in the Environment register; that is, the system data segment will be accessed even if the DS bit indicates user data.

Use of the SG-relative addressing mode is restricted to privileged mode only, and thus can be used only by callable and privileged procedures. Such procedures, when they operate on the user's data stack, are able to load, store, move, compare, and scan data, using both the user data segment and the system data segment.

Indirect addressing and indexing are both permitted with the SG-relative addressing mode.

SG-relative addressing mode is specified by 110 in the mode field of the memory reference instruction. The first example shows a direct reference to SG[4], as indicated by the value in the displacement field. The second example shows an indirect reference to SG[42176], through the direct location SG[13]. (Addresses are in decimal.) The X field specifies an index register, if used.



**Figure 6-17. Direct and Indirect Examples of SG Addressing**

VST282.vsd

# Basic Characteristics of Procedures

**Procedures** are the fundamental building blocks of programs. They are compiled in contiguous locations in a program file, and when the program is dispatched into execution as a process, the procedures appear in a code segment of virtual memory as shown in [Figure 6-18](#).

A procedure is a functional block of instructions that, when called into execution, performs a specific operation. It consists of a block of TNS instruction codes and program constants. A procedure can call another procedure within its own segment, or it can make an external call to a procedure in another segment of the same space or in any other available code space (UC, UL, SC, or SL).

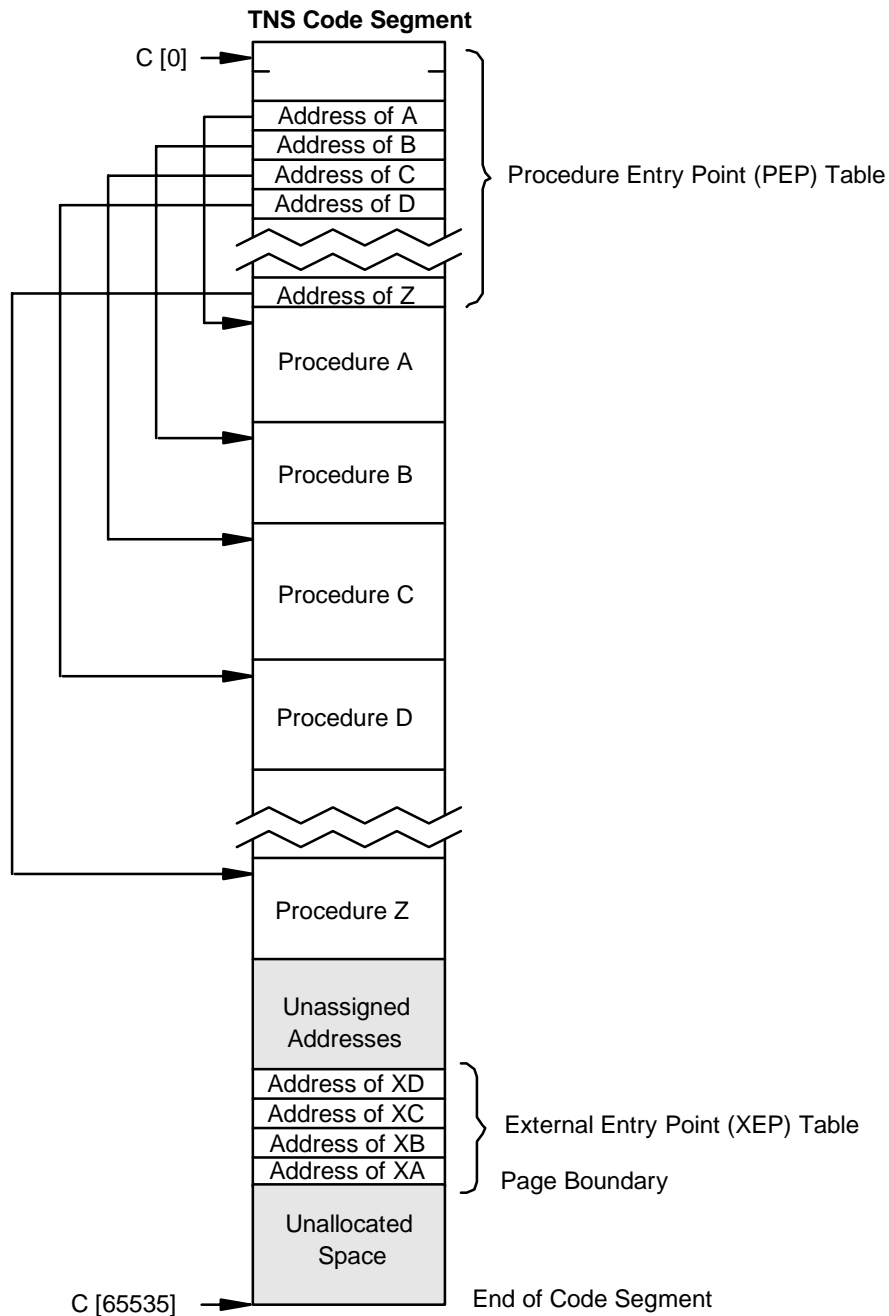
When one procedure calls another, the caller's environment is automatically saved before the other procedure begins, and the caller's environment is restored when the called procedure finishes. Parameters (or arguments) can be passed to the called procedure for evaluation, either as actual operands or as addresses of operands. Also, a procedure can return a value (such as the result of a computation) to its caller.

The address of the first instruction to be executed in a procedure is called the **entry point**. (A procedure can have multiple entry points.) All entry points for all procedures in a segment are located in a table, accessible by the hardware, called the **procedure entry point (PEP) table**. The PEP table itself is located at the beginning of each code segment.

The **external entry point (XEP) table**, also shown in the illustration, exists in each segment and is used when calling procedures in other code segments. (It is discussed in the topic [Calling External Procedures](#) on page 6-64.) This table ends on a page boundary, with entries consecutively assigned backward toward the end of code, using the first available space that fits (either on the same page as the end of code or on a separate page).

A procedure itself can contain one or more **subprocedures**. A subprocedure is similar to a procedure but can be called only by the procedure that contains it or by another subprocedure contained within the procedure. However, it has no entry in the PEP table and is implemented by a simple branch and return instruction pair (BSUB and RSUB). It executes only in the mode of its caller (nonprivileged or privileged).

Procedures are assigned a *callability* attribute. The attribute specifies whether or not the caller must already be executing in privileged mode and whether or not the called procedure executes in privileged mode. Attributes are considered next.

**Figure 6-18. Layout of Procedure Code in a TNS Code Segment**

VST283.vsd

# Procedure Attributes

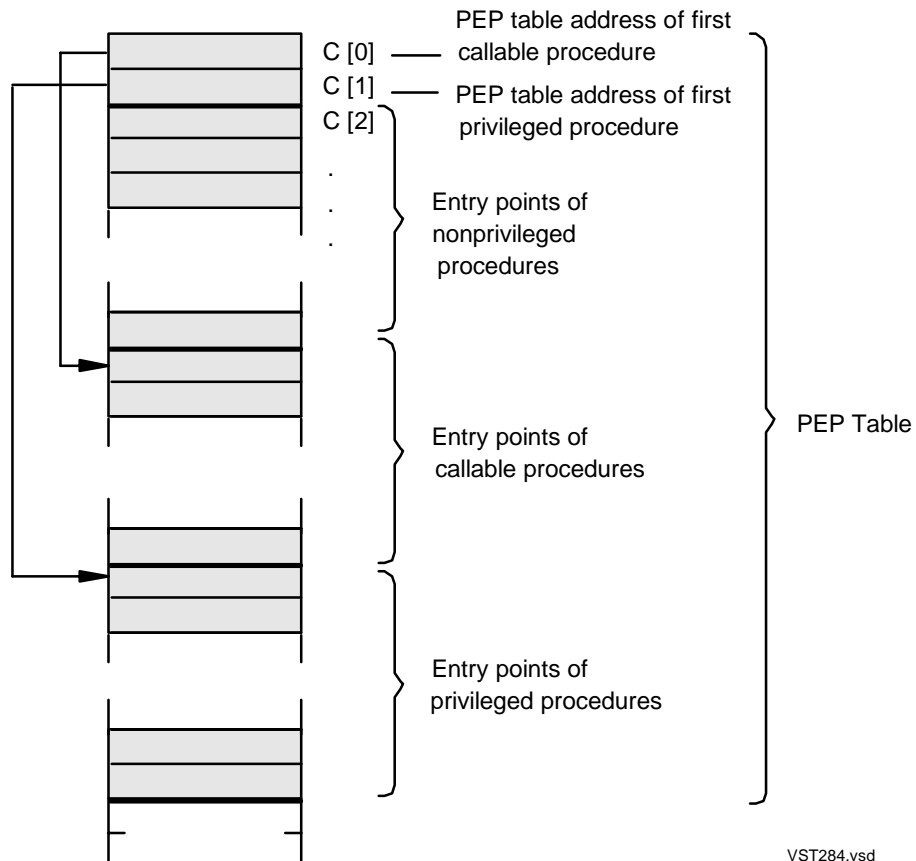
Some procedures use privileged instructions in performing certain operations. Those procedures can switch the processor into privileged mode (and thus be able to execute the privileged instructions) only if they have the required **procedure attribute**. Every procedure has one of the following attributes:

- |               |  |
|---------------|--|
| Nonprivileged | Procedures having this attribute can be called by any procedure. They execute in the same mode (privileged or nonprivileged) as the calling procedure. This is the attribute typically given to procedures in an application program.  |
| Callable      | Procedures having this attribute can also be called by any procedure, but they execute in privileged mode (that is, PRIV = 1 in the Environment register). The caller's mode is restored when a callable procedure exits. This attribute is typically assigned only to operating system procedures. It is used so that a controlled, secure interface exists between a nonprivileged application program and the privileged operating system.  |
| Privileged    | Privileged procedures execute in privileged mode and are callable only by procedures currently executing in privileged mode. An attempt by a nonprivileged procedure to call a privileged procedure results in one of several possible traps when the first privileged instruction or operation is encountered. This attribute should be used only by the operating system. It is typically used when the procedure uses privileged instructions which, if used improperly, might have an adverse effect on processor or system operation. |

In the procedure entry point (PEP) table, procedure entry points are grouped according to attribute. There are three groups: the first is all nonprivileged procedures, the second is all callable procedures, and the last is all privileged procedures. This grouping is shown in [Figure 6-19](#).

Because application programs rarely contain privileged code, PEP tables in user code and user library segments (UC and UL) generally have only the first group—that is, all procedures are nonprivileged. However, system library segments (SL) do have all three groups. System code segments (SC) have only the callable and privileged groups.

The first two words (TNS words) in the PEP table, C[0:1], describe where the callable and privileged entry points begin in the PEP table. Specifically, C[0] is the address of the first PEP entry for a callable procedure, and C[1] is the address of the first PEP table entry for a privileged procedure. These words are used to check whether a nonprivileged caller is attempting to invoke a privileged procedure. In the case where all procedures are nonprivileged, both of these addresses point just past the end of the PEP table (plus 1), so that any reference beyond the end of this table is invalid (invokes the privileged mode trap).

**Figure 6-19. Procedure Entry Points Are Grouped by Attribute in the PEP Table**

## Defining the Procedure's Data

As previously explained, the first half of the user data segment contains the process's global variables and TNS user data stack. The left part of [Figure 6-20](#) illustrates the overall organization of the user data segment.

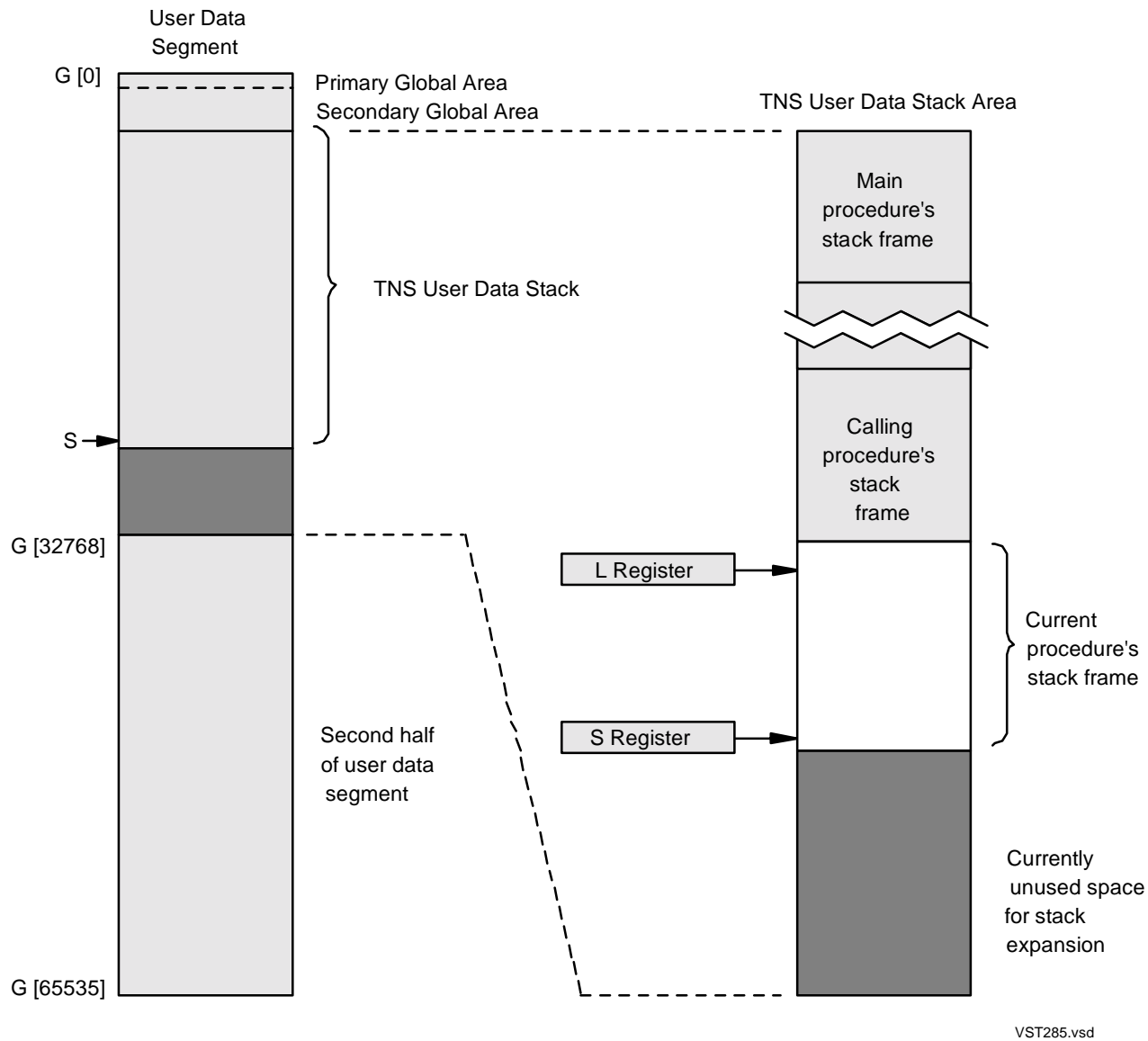
Variables in the **global data** area are addressable by any instruction in the TNS procedure. The addressing base of the global area is defined as G[0], and individual locations in this area are referenced with G-relative addresses. The primary global area exists within the first 256 words of the segment. This area is accessed with direct addressing. It can contain indirect addresses to anywhere in the segment, but they often point into an area of secondary global data beginning at a location immediately following the last primary variable. The secondary global data area can be much larger than the primary area.

The user data stack (or TNS stack) is where the manipulation of memory data by TNS instructions takes place. When a procedure is called into execution, that invocation is allocated its own temporary storage area on the stack, called a **stack frame**. This stack frame, which consists of memory data that the procedure is currently using, is known only to that invocation of the procedure and is logically separate from the stack frames of other procedures.

At any moment many frames may be in existence, one immediately following another. The first frame on the stack is that of the program's main procedure. The base of the stack frame is defined by the 16-bit **L register**. The L (for Local) register contains the G-relative address base of the procedure's local data and is defined as L[0]. Elements following (and preceding) L[0] can be addressed with positive or negative offsets.

The current **top-of-stack** location is defined by the 16-bit **S register**, which contains the G-relative address of the last word currently defined in the user data stack. This address is defined as S[0], and it serves as an addressing base for a top-of-stack area called the **sublocal area**. This sublocal area consists of up to 32 word locations including and preceding S[0]; it is typically used by subprocedures. Data in the sublocal area is known only to the currently executing subprocedure.

During the execution of a procedure, the address in the S register advances as elements are moved from the register stack to the top of the user data stack, and recedes as elements are moved from the top of the user data stack to the register stack.

**Figure 6-20. Procedure Data Consists of Global Areas and Stack Frame**

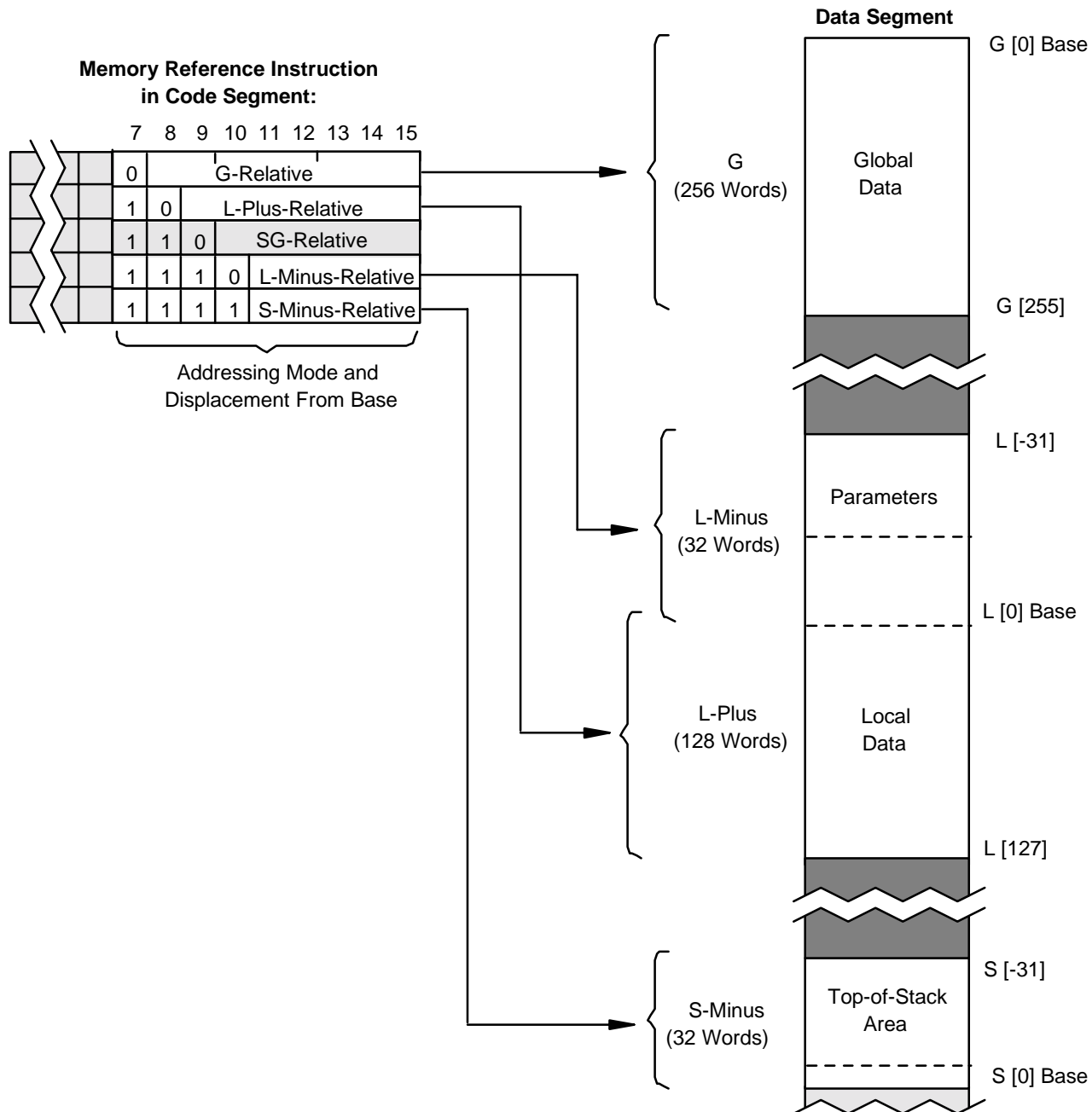
# Data Segment Addressing Modes

All direct addressing in the data segment is relative to one of the three addressing bases: G, L, or S. Memory reference instructions for data contain a 9-bit address field for specifying one of the three addressing bases and a relative displacement from that base. Four addressing modes are provided for addressing relative to these bases. (A fifth mode, SG-relative addressing, exists for addressing relative to a base in the system data segment; that mode is described earlier in this section.)

For addressing in the user data segment, the addressing modes are **G-relative**, **L-plus-relative**, **L-minus-relative**, and **S-minus-relative**. The following paragraphs describe these four addressing modes. Each is illustrated in [Figure 6-21](#). The illustration also defines the bit patterns used in instructions that use these modes.

G-Relative Mode	This mode addresses the first 256 locations in the global area, G[0:255]. The G-relative mode is indicated when bit 7 of a memory reference instruction is equal to 0; bits 8:15 specify a positive word displacement from G[0].
L-Plus-Relative Mode	This mode addresses the first 128 words of a procedure's local data area, L[0:127]. The L-plus-relative mode is indicated when bits 7:8 of a memory reference instruction are equal to 10 (binary); bits 9:15 specify a positive word displacement from the current L[0]. The processor calculates a G-relative address by adding bits 9:15 to the contents of the L register.
L-Minus-Relative Mode	This mode addresses the 32 words just below and including the word pointed to by the current L register setting, L[−31:0]. (This area is used for accessing procedure parameters.) The L-minus-relative addressing mode is indicated when bits 7:10 of a memory reference instruction are equal to 1110 (binary); bits 11:15 are a negative word displacement from the current L[0]. The processor calculates a G-relative address by subtracting bits 11:15 from the contents of the L register.
S-Minus-Relative Mode	This mode addresses the 32 words just below and including the current top-of-stack word, S[−31:0]. (This area is used for a subprocedure's sublocal data and for temporary storage of the register stack contents by the PUSH and POP instructions.) The S-minus-relative mode is indicated when bits 7:10 of a memory reference instruction are equal to 1111 (binary); bits 11:15 are a negative word displacement from the current S[0]. The processor calculates a G-relative address by subtracting bits 11:15 from the contents of the S register.



**Figure 6-21. There Are Four Data Segment Addressing Modes**

VST286.vsd

## Operations at the Procedure's Top-of-Stack

The top-of-stack area can be addressed implicitly through use of the PUSH and POP instructions. Implicit addressing means that it is not necessary to specify a mode or a register. The S register is implied by both instructions; all operations take place at the top of the memory stack. A word count is used rather than a displacement value.

The **PUSH instruction** is used to store some or all of the register stack contents to the top of the memory stack. When a PUSH instruction is executed, the S register setting is incremented by the number of words pushed. The **POP instruction** is used to load one or more of the registers with contents from the top of the memory stack. When the POP instruction is executed, the loaded words are deleted from the memory stack by decrementing the S register setting accordingly.

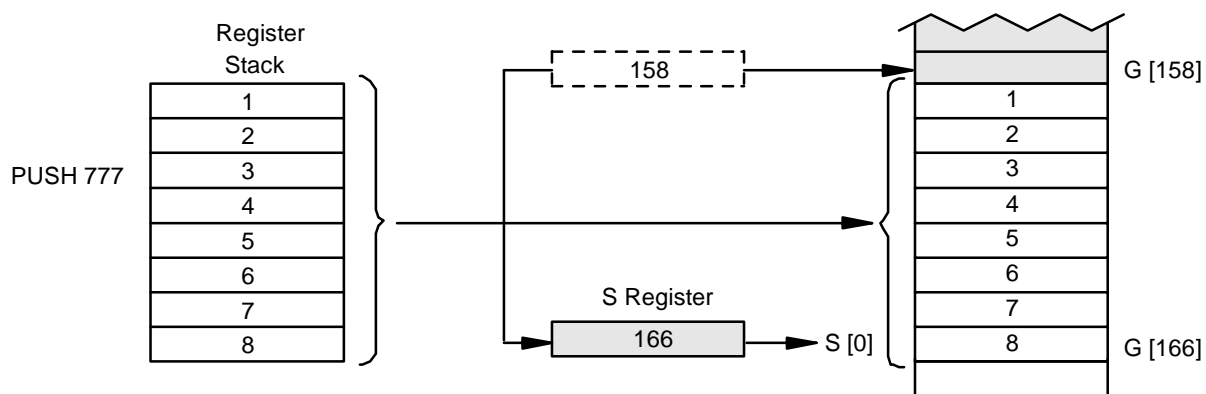
[Figure 6-22](#) illustrates both the PUSH and the POP instructions. In both examples, the register stack is on the left and the memory stack is on the right. The S register settings are shown in the middle. The instruction as written, including a three-digit octal number, is listed at the extreme left. Each digit of the octal number has the following meanings, in order from most to least significant: (1) the new RP value following execution, (2) the last register to be stored or loaded, and (3) the count of registers to be stored or loaded (minus 1). The latter two of these values have the effect of specifying how many registers are affected and which ones.

In the PUSH example, all eight registers are to be stored. (They contain consecutive numbers to illustrate storage order.) The number 777 indicates that the count is 8 (specified as 8 minus 1, or 7), that the last register to be stored is R[7], and that RP will indicate 7 at the end of the instruction. As indicated in the S register settings, S advances the top-of-stack from G[158] to G[166]. The values are stored in consecutive memory locations, with the content of R[7] (the value 8) as the last value to be stored.

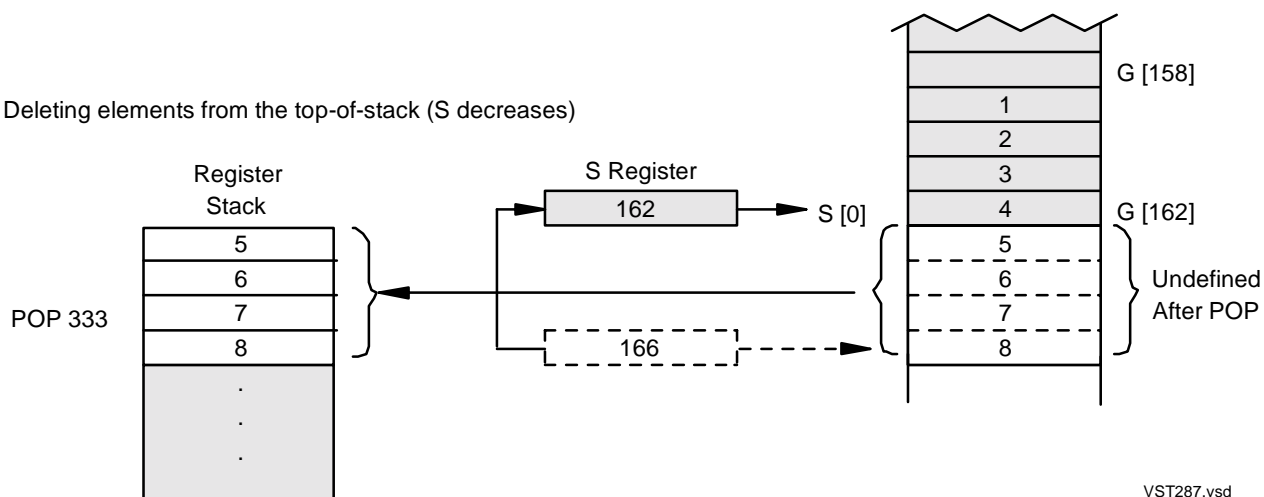
In the POP example, four registers are loaded from the top of the memory stack. The count is 4 (given as 4 minus 1, or 3), and the last register loaded is R[3]. After the four elements are loaded, the value 8 is in R[3] and the value 5 is in R[0]. The S register decrements the top-of-stack from 166 to 162 to delete the four elements from the memory stack.

**Figure 6-22. PUSH and POP Instructions Add and Delete Stack Elements**

Adding elements to the top-of-stack (S increases)



Deleting elements from the top-of-stack (S decreases)



VST287.vsd

# Overview of Procedure Call and Exit

Procedures are invoked using **procedure call instructions**—PCAL to a procedure within the same code segment, XCAL to a procedure in some other code segment, or DPCL to an indirectly specified target procedure. An example of a procedure call and exit is shown in [Figure 6-23](#). This example assumes the called procedure is in the same segment as the caller, UC.2; therefore, the PCAL instruction is used.

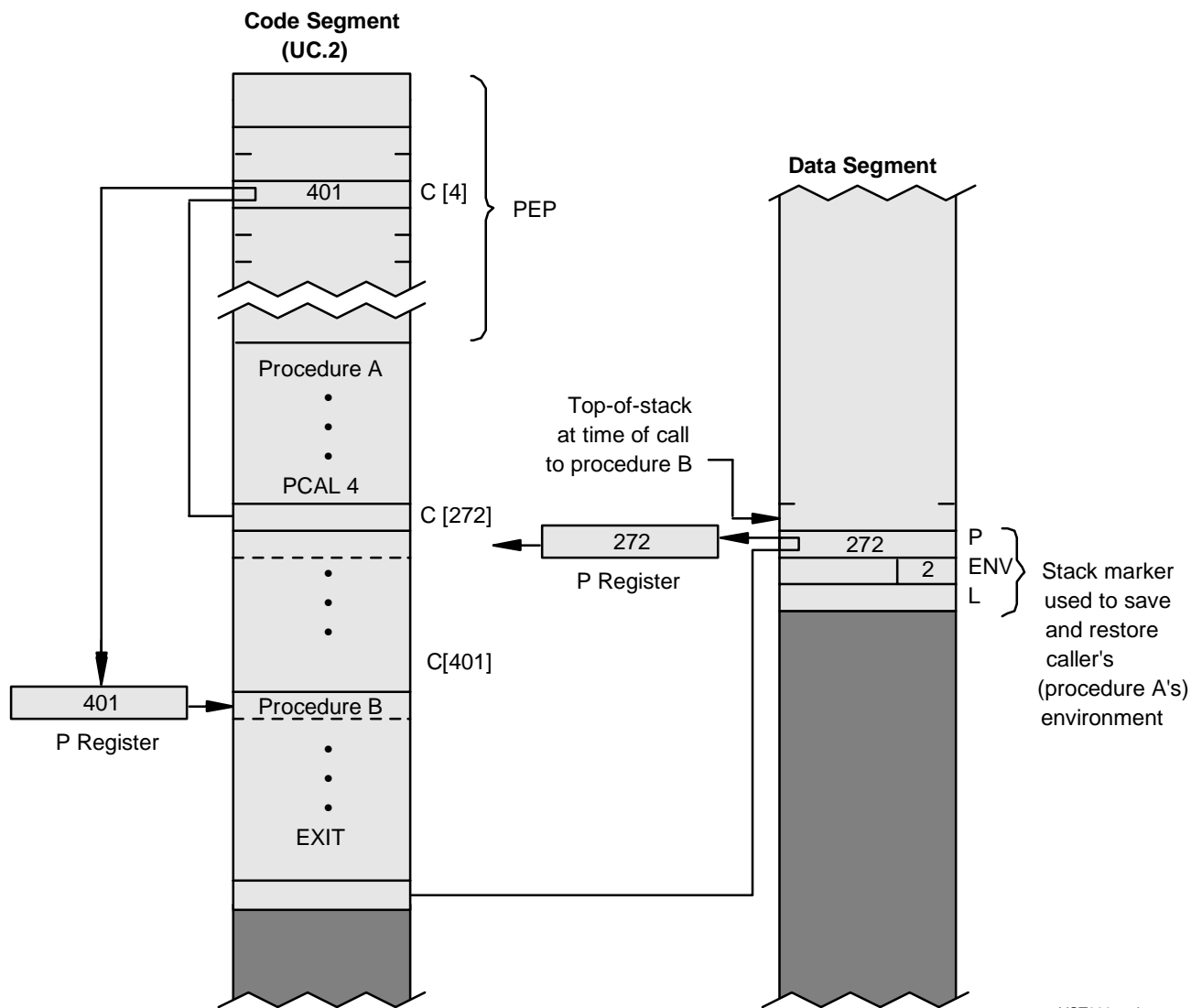
The beginning action of either of these instructions (PCAL, XCAL, or DPCL) is to save the caller's **environment**. A procedure's environment is the code and data that it is currently using. For code, the environment specification needs to include complete segment identification (which code or library space, and which segment within that space). For data, the environment is its stack frame. Both aspects of environment preservation are accomplished by the use of a three-word **stack marker** to separate stack frames within the memory stack. The data inside the marker provides the restoration information, specifically the address of the instruction following the call (P register value), the caller's Environment register setting, and the L register setting of the caller. The saved version of the Environment register is slightly modified to include a space ID index; note the space ID index value of 2 in the example.

Once the caller's environment is preserved, a new environment is set up for the called procedure. First, the L register is set to the value in the adjusted S register. This defines the base of a new stack frame for the called procedure's local data area.

The call instruction (PCAL in this case) then accesses the entry in the PEP (procedure entry point) table corresponding to the procedure being called. The called procedure's callability attributes are checked first, then the address in the PEP entry is placed in the P register so that the next instruction executed is the one at the called procedure's entry point. The called procedure now executes, using its own stack frame on the data stack for its local data.

The last instruction that a procedure executes is an EXIT instruction. The EXIT instruction returns control to the caller. In brief overview, this return of control is accomplished by restoring the caller's L and S register settings and by setting into the P register the return address (that is, the address of the instruction following the original call instruction). The caller's Environment register setting also is restored—except for the Condition Code (CC) and Register Pointer (RP) fields, which are left as is; that is because these fields in the stack marker copy of the Environment register were used to save the space ID index.

The next two topics consider in greater detail the operations of the PCAL and EXIT instructions. The special case of calling external procedures using the XCAL instruction is described later in this section.

**Figure 6-23. Example of a Procedure Call and Exit**

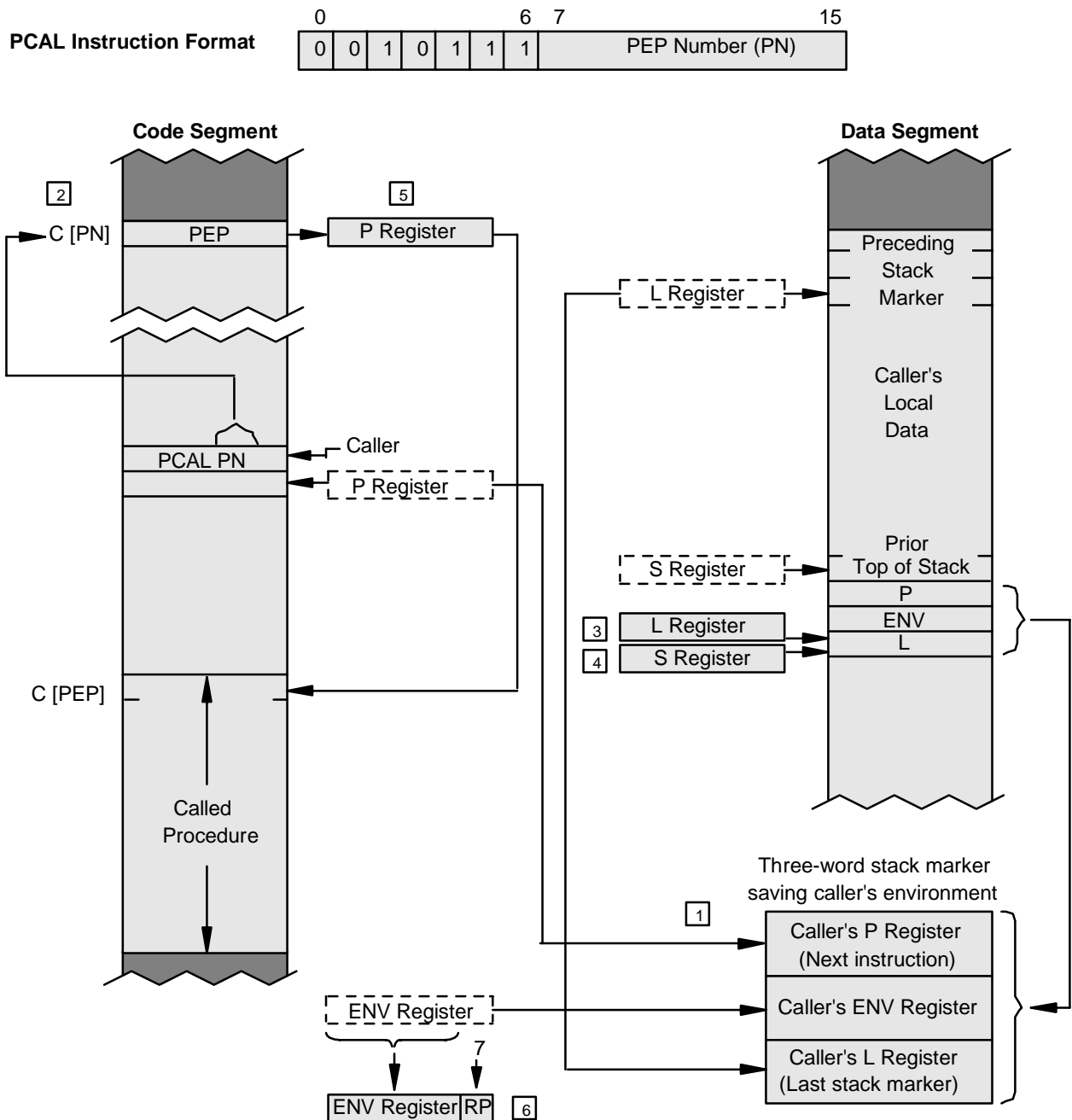
# Actions of the PCAL Instruction

The following steps are performed when a Procedure Call (PCAL) instruction is executed. The step numbers refer to [Figure 6-24](#). Note that before PCAL executes, the program must push the procedure parameters (and the mask word or words, for procedures with a variable number of parameters) onto the memory stack.

1. The caller's environment is saved in a three-word stack marker, which is pushed onto the top-of-stack location, as indicated by the address in the S register. The stack marker contains the following information:
  - The current P register setting (address of the instruction following PCAL).
  - The current Environment register setting. The stored copy contains the complete segment identification of the caller's segment, even though this call and return is within a single segment.
  - The current L register setting (the beginning of the caller's local data area).
2. If the calling procedure is not executing in privileged mode, the "callability" attribute of the procedure being called is checked.

First, the PEP number field of the PCAL instruction is compared with the entry in C[0]. If the PEP number is less than the C[0] entry, this is a call to a nonprivileged procedure and no special action is taken. However, if the PEP number is greater than or equal to the C[0] entry, this is a call to a callable or privileged procedure, so a second check is made. The PEP number is compared with the entry in C[1]. If the number is greater than or equal to C[1], this is a call to a privileged procedure and an instruction failure trap occurs. Otherwise, this is a call to a callable procedure, so the PRIV bit is set.

3. The S and L registers are set with the G[0]-relative address of the new top-of-stack location (the third word of the stack marker). The new L register setting defines the base of the local area for the procedure being called.
4. The new S register setting is tested for being an address within the memory stack area, G[0:32767]. If the value is greater than 32,767, control is transferred to the operating system's stack overflow trap (aborting the PCAL instruction).
5. The C-relative address of the procedure being called is obtained from the PEP table entry pointed to by the PEP number field in the PCAL instruction. This address is put in the P register so that the next instruction executed will be the first instruction of the called procedure.
6. Finally, RP is given an initial value of 7 (stack empty).

**Figure 6-24. Sequence of Events Performed by a PCAL Instruction**

VST289.vsd

## Actions of the EXIT Instruction

The EXIT instruction uses the three-word stack marker to restore the caller's environment. The sequence is as follows, with reference to [Figure 6-25](#). (For simplicity, and continuity with the preceding PCAL description, this sequence assumes the return is from a procedure that was called with PCAL rather than XCAL.)

1. The S register setting is moved below the local area, the stack marker, and any parameters to the exiting procedure.

The S<sup>^</sup>decrement value (which is specified in the EXIT instruction) is subtracted from the current L register setting and placed in the S register. The value of S<sup>^</sup>decrement is 3 (for the stack marker) plus the number of words of parameter and mask information passed to the exiting procedure.

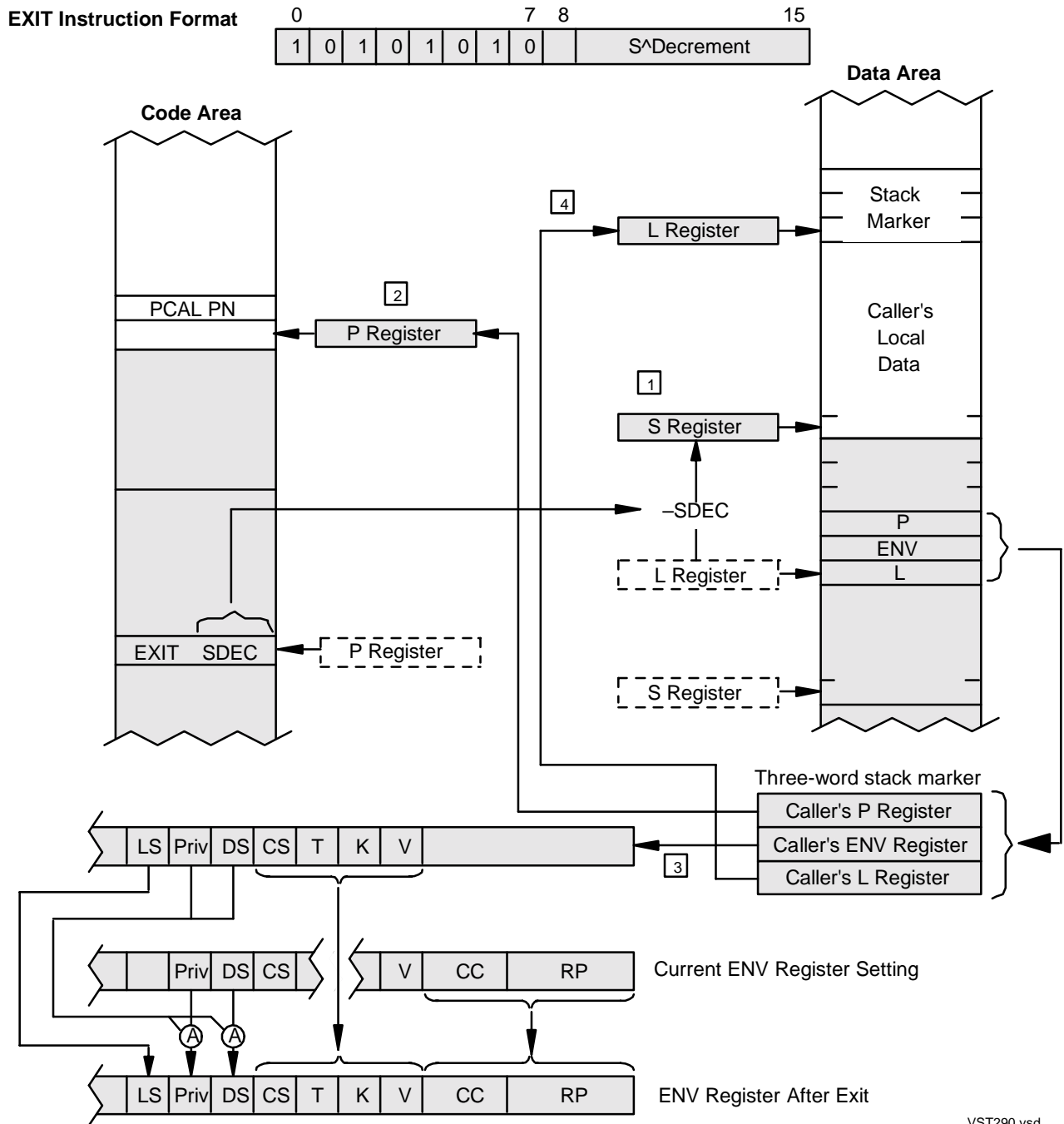
2. The P register is set with the P register value saved in the stack marker at L[−2]. The next instruction to be executed will be the one following the PCAL instruction.
3. The Environment register is restored from a combination of the current Environment register setting and the Environment register value saved in the register stack at L[−1].

The mode (privileged or nonprivileged) and data area bits are reestablished to be the lesser of the caller's and the current settings. This ensures that a nonprivileged user cannot exit with privileged capability. The caller's CS (code space), LS (library space), T (traps), V (overflow), and K (carry) bits are reestablished from L[−1]. Z and N (Condition Code) are left at their current settings to reflect the results of the call. RP is left at its current setting so that a value in the register stack can be returned to the caller as a function result value.

4. The L register is restored from the L register value saved in the stack marker at L[0]. This moves L back to point to the preceding stack marker, thereby reestablishing the preceding local data area.

The instruction following the PCAL instruction then executes.



**Figure 6-25. Sequence of Events Performed by an EXIT Instruction**

VST290.vsd

## A Procedure's Local Variables

Unlike the global data area, which exists at all times during the life of the process, the local data area for a procedure exists only during the time between the procedure's being called and its exiting.

One of the subdivisions of the local data area is used specifically for **local variables**. These are found in a certain number of locations (up to 127) immediately succeeding the stack marker. Because the L register specifies the ending location of the stack marker, the local variables are addressed by positive displacements from the L register setting.

As indicated in [Figure 6-26](#), the L-plus-relative addressing mode is specified by the bit pattern of 10 in bits 7 and 8 of the instruction word. The seven-bit field consisting of bits 9 through 15 specifies the word displacement from L. Either direct or indirect addressing can be used, indicating that any location in the local variables area can contain either a value (direct) or an address (indirect).

The local variables are allocated and initialized by instructions at the start of a procedure's code. Thus, a procedure can be called any number of times (and in fact can call itself), and each call generates a fresh copy of the procedure's local variables area.

In the example illustrated in the figure, assume there are three local variables declared in a TAL source program: "i" is a one-word uninitialized variable, "j" is a one-word variable initialized with the value 5, and "k" is an indirectly addressed array variable consisting of 32 words. The instructions to generate these variables are:

```

ADDS      +001      ! Add to S              (make room for i)
LDI       +005      ! Load Immediate      (initialize j)
LADR      L+004      ! Load Address         and
PUSH      711        ! PUSH to Memory      pointer to k)
ADDS      +040      ! Add to S              (allocate space for k)

```

The first ADDS instruction increments the S register setting by 1. This allocates one word for the variable "i".

The LDI instruction puts the initialization value for "j" (5) on the top of the register stack.

The LADR instruction calculates the G-relative address of the first word of the indirect array "k" and puts the address on the top of the register stack.

The PUSH instruction performs two functions: (1) it puts the initialization value given in "j" and the address of the array "k" into L[2] and L[3] of the process's stack, respectively, and (2) it increments the S register setting by 2 to allocate the two words needed for "j" and the address pointer to "k".

The last ADDS instruction increments the S register setting by 32 (octal 40). This allocates 32 words for the indirect array "k".

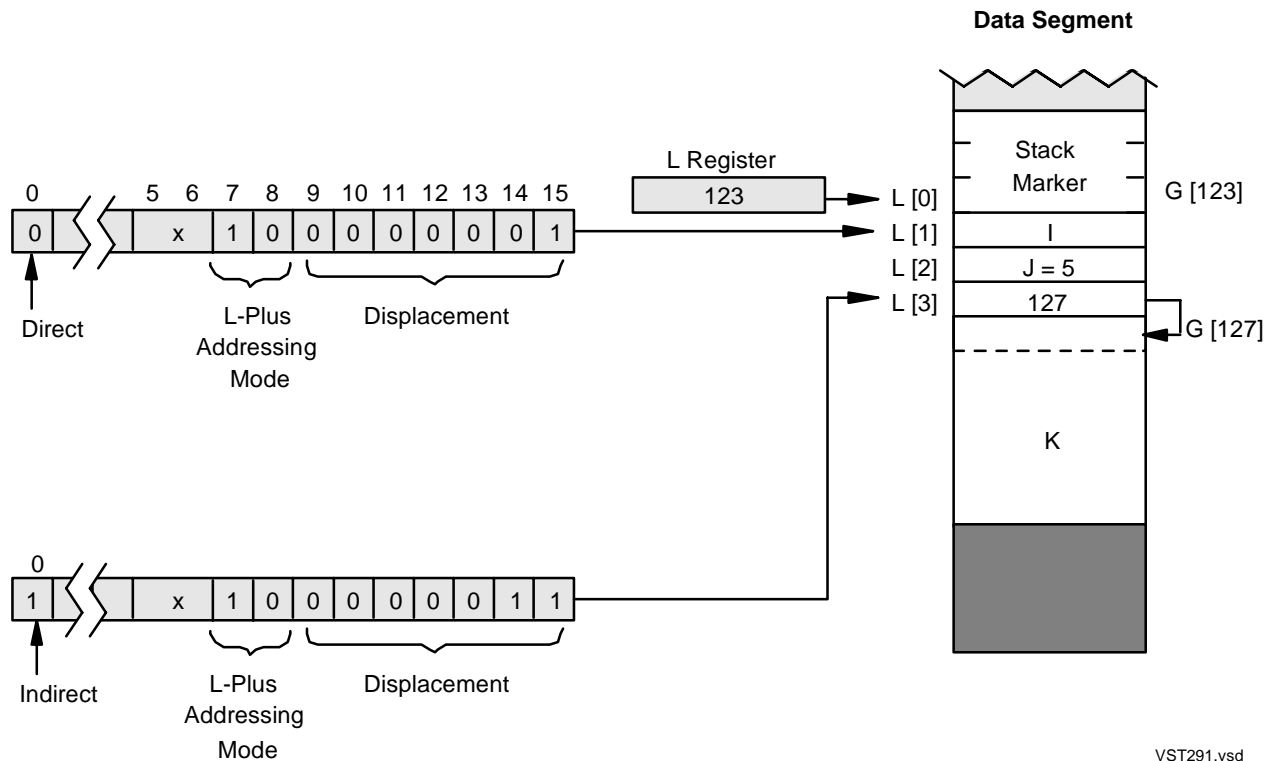
Following the generation of the local variables, this area consists of:

```

L[1]      = i
L[2]      = j (initialized with a value of 5)
L[3]      = an address pointer to the array "k"
L[4:35]   = the array "k"
  
```

Once allocated, data in the local area is addressed relative to the current L register setting using the L-plus-relative addressing mode. As illustrated, this mode can access local data directly or can use the direct address as an address pointer (indexing is also permitted).

**Figure 6-26. Defining and Accessing a Procedure's Local Variables**



VST291.vsd

# Passing Parameters to a Called Procedure

Parameters are passed to a procedure in the top-of-stack area (memory stack). Naturally, there must be coordination between the caller and the called procedure when passing parameters. The caller must know the order in which a procedure expects parameters, and whether a parameter is to be an actual operand (called a **value parameter**) or an address pointer (called a **reference parameter**).

Before the caller invokes a procedure, the parameters are prepared in the register stack. The actual operands (for value parameters) and the addresses of operands (for reference parameters) are loaded into the register stack in the order required by the procedure being called. The address of a reference parameter is obtained by the execution of a LADR (load address) instruction. The parameters that have been prepared in the register stack are loaded on the top of the memory stack by executing a PUSH instruction (which increments the S register accordingly).

The example shown in [Figure 6-27](#) illustrates the instructions used to prepare the top of the memory stack area for parameter passing. The TAL procedure being called is of the form:

```
PROC b (p1,p2);
      INT p1,.p2;
```

Parameter “p1” is a value parameter; therefore, the procedure expects an actual value to be passed. Parameter “p2” is a reference parameter, and, therefore, the procedure expects the G-relative address of a variable to be passed.

The call being made from procedure A is:

```
CALL b (j,i);
```

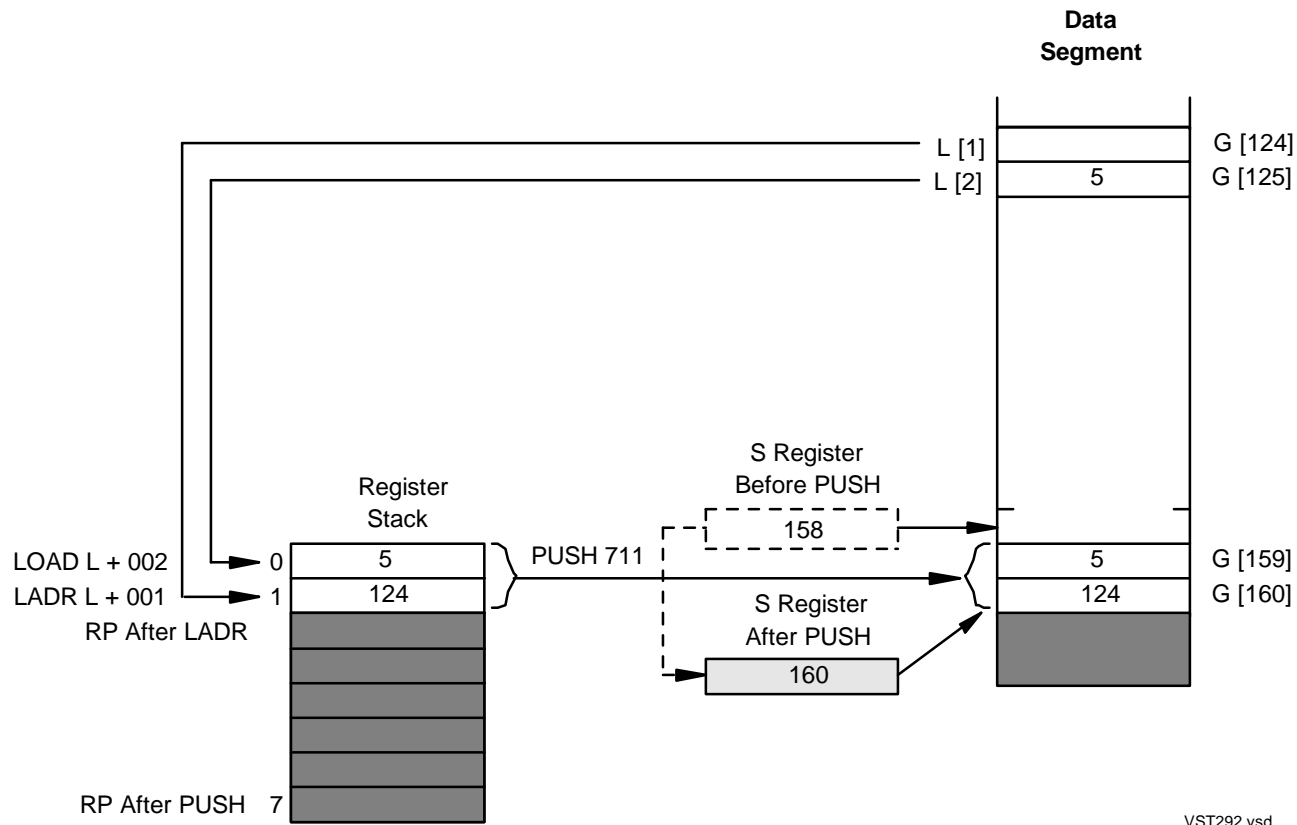
The instructions to pass these two parameters are:

```
LOAD L +002
LADR L +001
PUSH 711
```

The LOAD instruction puts the contents of the variable “j” (the value 5) on the top of the register stack. (This is the parameter passed as “p1”, a value parameter, to procedure B.)

The LADR instruction calculates the G-relative address of the variable “i” and puts the address on the top of the register stack. (This is the parameter passed as “p2”, a reference parameter, to procedure B.)

The PUSH instruction places the two parameters from the register stack on the top of the memory stack and increments the S register setting by 2.

**Figure 6-27. Example of Passing Parameters to a Called Procedure**

VST292.vsd

## Accessing Parameters in the Called Procedure

Procedure parameters are accessed by using the L-minus-relative addressing mode. This mode provides access to the 32 locations just below and including the current L register setting ( $L[-31:0]$ ). Subtracting the three words used for the stack marker, this leaves 29 words that can be directly addressed as parameters.

If value parameters are passed, the parameter location is addressed directly (indirect bit of a memory reference instruction = 0); if reference parameters are passed, the parameter location is used as an indirect address (indirect bit = 1). Indexing is permitted in either mode.

[Figure 6-28](#) shows examples of both value and reference parameter access. In both cases, the L-minus mode is specified by a binary 1110 in bits 7 through 10 of the instruction word. The five-bit displacement field provides for a 32-word offset (negative direction only).

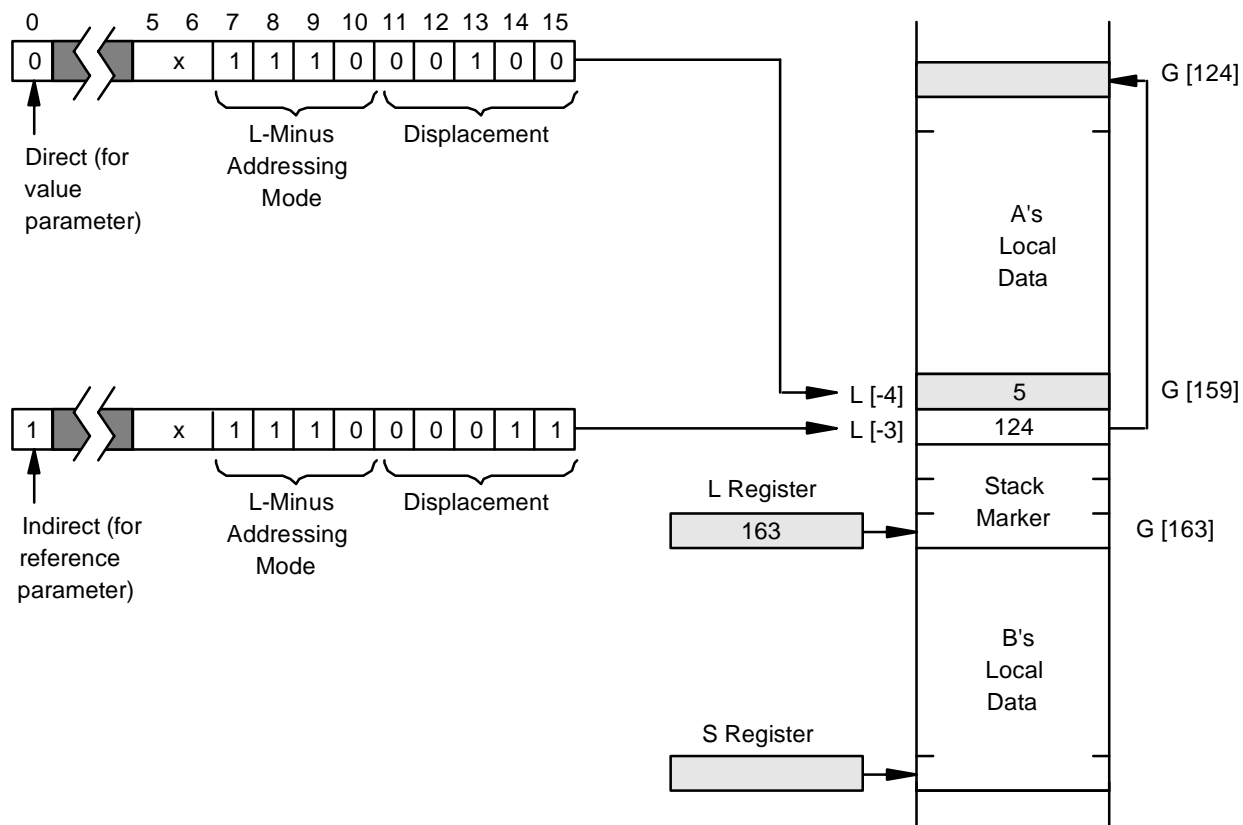
The example assumes, in both cases, that the L register currently is pointing at  $G[163]$ . That means that the data beginning at  $G[164]$  belongs to the called procedure (B). The data preceding the stack marker ( $G[160]$  and lower) is the area for passing parameters and can be accessed both by the calling procedure (A) and by the called procedure (B). Any data within the L-minus addressing range (29 words) is directly accessible to the called procedure for passed parameters.

In the first example (direct addressing), the displacement is  $-4$  from L, and so the content of  $G[159]$  is a value parameter that the called procedure can access and use.

In the second example (indirect addressing), the displacement is  $-3$  from L, and so the content of  $G[160]$  is a reference parameter pointing to location  $G[124]$ . That location (farther back into the calling procedure's data) contains the value that the called procedure can use as the passed parameter.

In the first example, the displacement of  $-4$  from L addresses a location two words prior to the stack marker; direct addressing makes the content a value parameter. In the second example, the displacement of  $-3$  from L addresses the location immediately preceding the stack marker; indirect addressing makes the content a reference value that points further back for the parameter, to  $G[124]$ .

**Figure 6-28. Examples of Accessing Value Parameters and Reference Parameters**



VST293.vsd

# Saving the Stack Frame on a Call

When a procedure is called, a new stack frame is defined. This occurs because the address contained in the L register advances to point above the current local area (the caller's local area is then inaccessible by direct addressing). Conversely, when a procedure exits, the exiting procedure's stack frame is deleted (and the preceding stack frame made accessible again) because the address in the L register recedes back to its previous setting.

This and the following topics depict an example of memory stack operations from an initial state (that is, start of process execution) through a call to, and subsequent return from, a procedure. The purpose of this example is to show the action of the L and S registers as a procedure generates its local variables and prepares to call a procedure by passing parameters, how L and S are advanced when a procedure is called, and how L and S are restored when the return is made to the caller.

The following numbered steps refer to [Figure 6-29](#).

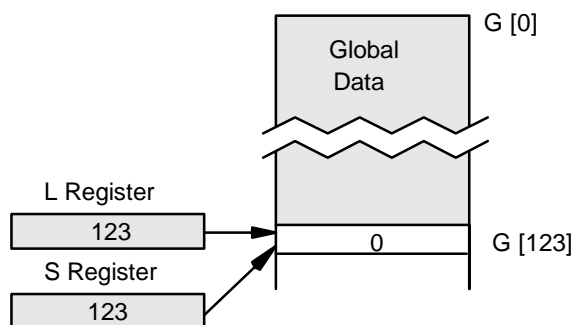
1. **Initial State.** After the operating system has loaded a program into memory but before the first instruction of the process executes, the following initial conditions are present: the process's global variables are initialized and present, and the L and S registers are set to the address of the word just past the global area. There are no local variables defined at this time.
2. **Procedure A allocates its local variables.** The first few instructions of a procedure allocate the procedure's local variables. As the local variables are allocated, the S register setting increases, defining a new upper limit to the procedure's stack frame. Note that the L register setting does not change.
3. **Procedure A prepares parameters for procedure B.** In preparation for calling the procedure B, the parameter words (two in this example) are placed on the top-of-stack location as indicated by the S register setting. The S register setting is increased by 2 to account for the parameters.
4. **A calls B.** After the parameters are loaded onto the memory stack, a procedure call instruction is executed; this could be a PCAL (Procedure Call), XCAL (External Procedure Call), or DPCL (Dynamic Procedure Call). Execution of the call instruction places a three-word stack marker at the current S register setting plus 1 (just above the parameters). The L and S registers are given a new setting; they both point to the third word of the stack marker. The new L register setting defines the start of B's local area. At this point, no local variables have yet been allocated for procedure B. (Note that A's local area, which is normally addressed relative to the L register, is no longer addressable by the L-plus addressing mode.)

(The next topic continues this sequence.)

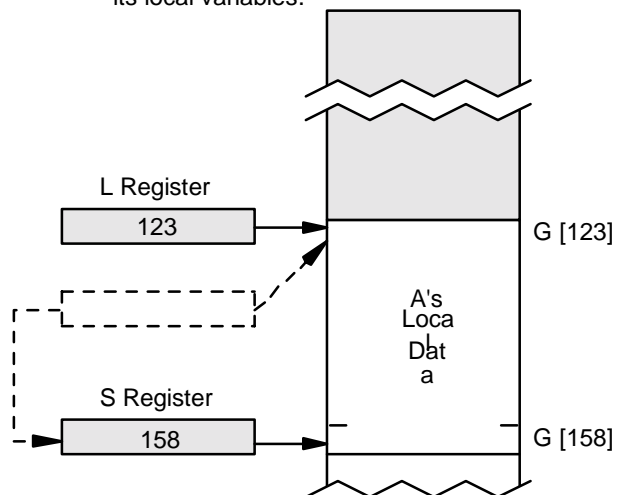


**Figure 6-29. Sequence of Events for Saving a Stack Frame**

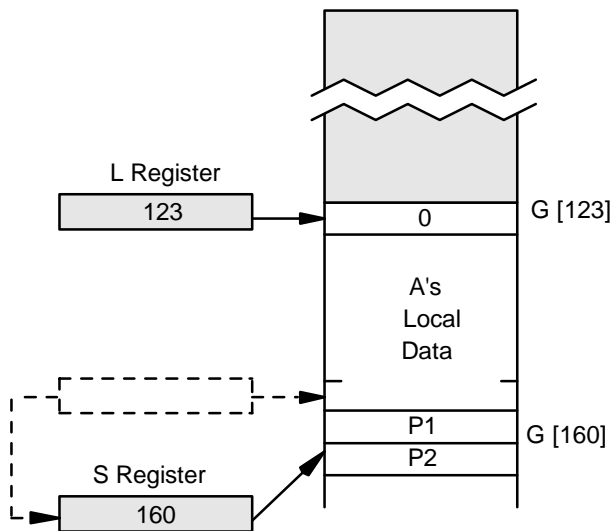
1 Initially (program starts)



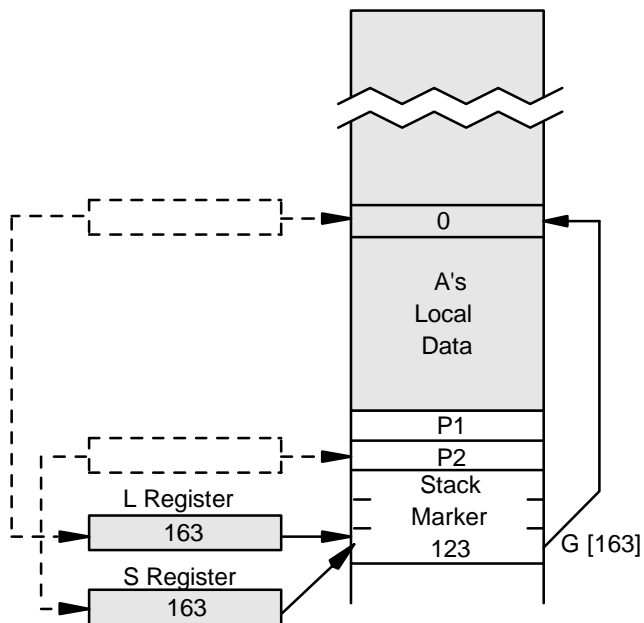
2 Procedure A generates its local variables.



3 Procedure A puts parameters on the stack in preparation for calling B.



4 Procedure A calls procedure B.



VST294.vsd

# Restoring a Stack Frame on Return From a Call

A called procedure assumes full control of the process's memory stack, accumulating and using the data in its own stack frame. When that procedure completes its operations and exits, the stack frame it was using is no longer needed and so is deleted. The method for deleting the stack frame is to restore the L and S register settings to those that were in effect in the calling procedure at the time the call was made.

The following sequence and [Figure 6-30](#) show how this restoration is accomplished. (This sequence is a continuation of the preceding topic.)

5. **Procedure B allocates its local variables.** In the same manner as procedure A did, procedure B generates its local variables. This increases the S register setting accordingly, so that the S register defines the new upper limit to B's stack frame.
6. **Procedure B exits back to procedure A.** When procedure B finishes, it executes an EXIT instruction to return to A. Execution of the EXIT instruction moves the L register setting back to the beginning of A's local area and moves the S register setting back to the top-of-stack location that was in effect before the parameters were loaded on the stack for B. The new S register setting is derived from the old L register setting, because the L register contains the pointer to the stack marker that divides the two stack frames. The deletion of the parameter words is accomplished by use of the "S^decrement" value in the EXIT instruction.

The S^decrement value specifies how many word locations the S register should be set back from the old L register setting. Because, at the very least, the old stack marker needs to be deleted, the S^decrement value should at least be 3. Any value over that number will also delete words that precede the stack marker. Because the called procedure knows how many words preceding the stack marker must be passed to it as parameters, the EXIT instruction can be made to delete those words along with the stack marker.

In the case illustrated, for the return to procedure A, the EXIT instruction is:

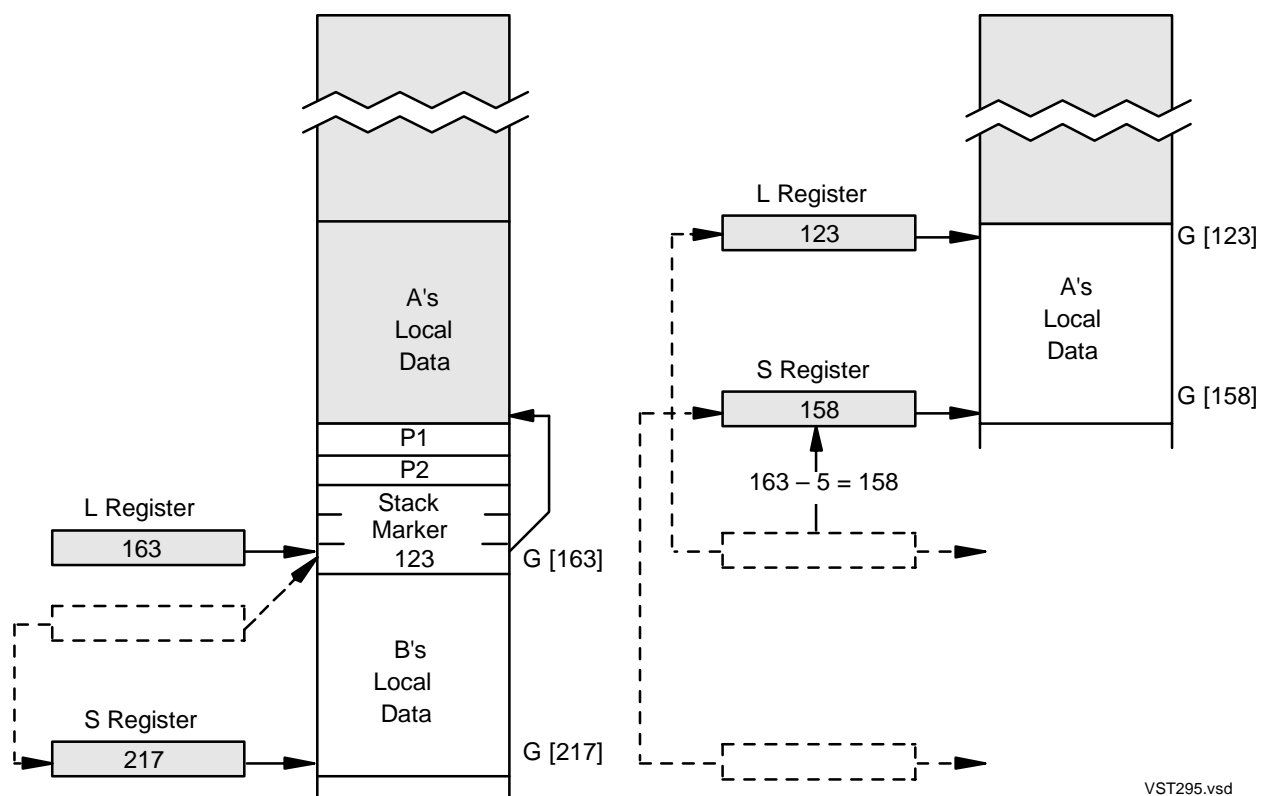
```
EXIT 5
```

This deletes the three-word stack marker, plus the two parameter words, from the top of the stack—in addition to all of B's local data. Procedure A is now back to where it was before the call was made.

**Figure 6-30. Sequence of Events for Restoring a Stack Frame**

5 Procedure B generates its local variables.

6 Procedure B exits back to A.



## Multiple Markers for Nested Calls

In examples shown previously, only one procedure call occurred and, therefore, only one stack marker was generated. However, in practice, there can be several stack markers (and stack frames) present in a memory stack at once. This occurs when a called procedure calls another procedure and that procedure calls still another procedure, and so on. Because one of the words in each marker (the saved L register) points back to the previous stack marker, the result is a “chain” of stack markers.

The nature of this chain of stack markers and the action of the L and S registers is such that the stack frames are positioned in chronological order, and returns are always made in the reverse order of the calls. The stack frames are made current again as the returns are made.

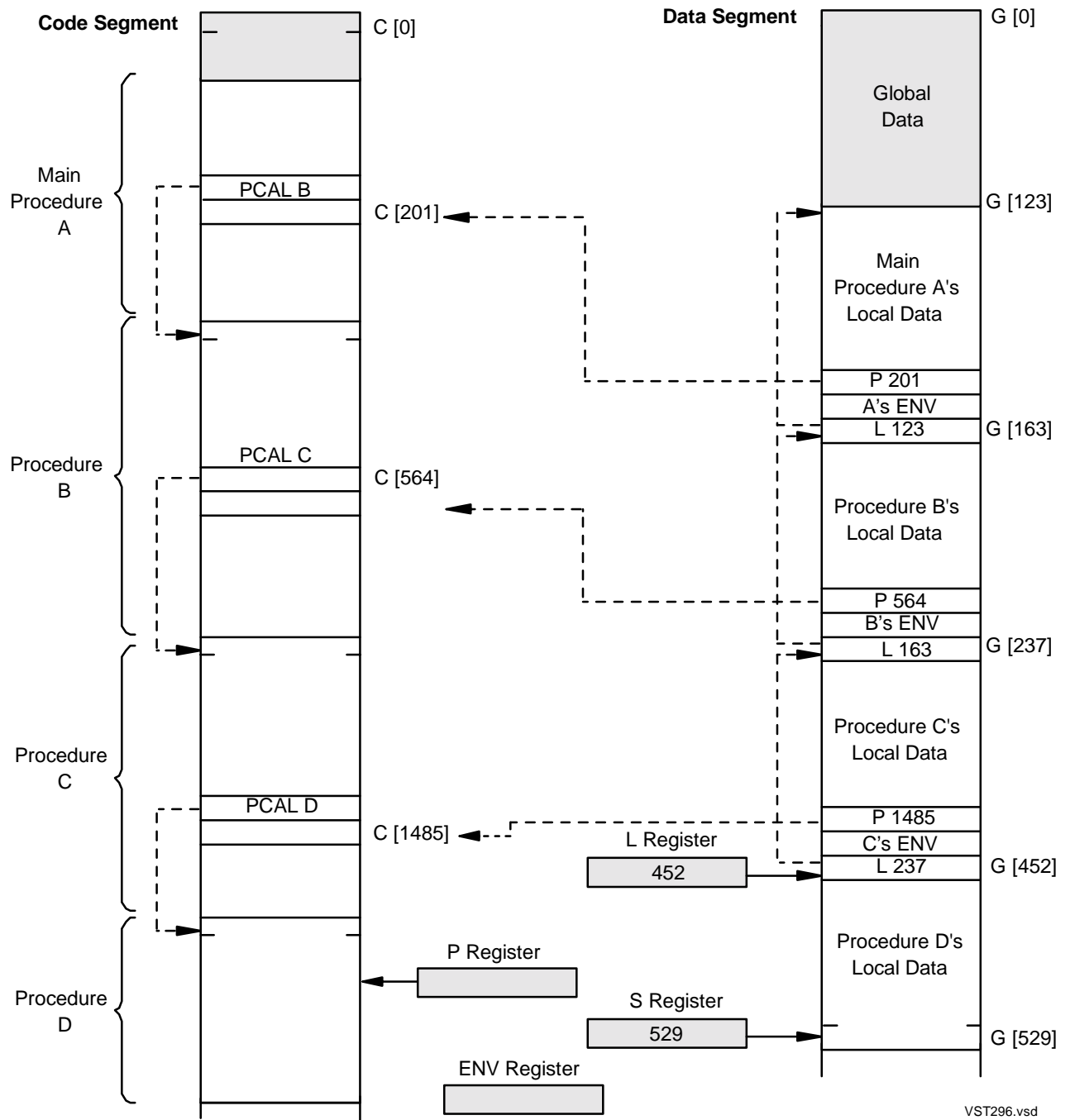
[Figure 6-31](#) shows the condition of a memory stack after the following calls have taken place:

```
In main procedure A, CALL B;
    In procedure B, CALL C;
        In procedure C, CALL D;
```

This sequence can be traced out in the fragment of code segment shown at the left side of the figure. Note the PCAL instruction in each procedure pointing to the entry point of the succeeding procedure. (For simplicity, calls are shown going to the next contiguous procedure in the segment; in practice, calls can jump from one extreme to the other in the code segment.) The procedure D is currently executing, as indicated by the P register setting.

Now note the chain of stack markers on the right side of the figure. The current L register setting is 452. At G[452] is the saved L value in the most recent stack marker. That value is 237, meaning that when procedure D exits back to C, the L register pointer will jump back to G[237]. Likewise, when C exits back to B, L will jump back to G[163], and when B exits back to A, L will jump back to G[123]. At the moment shown, however, the chain exists as an explicit chain of references going back to the base of the first stack frame on the stack.

Note also that the first word of each marker contains the P register value that tells where the preceding procedure must resume execution when the return is made. For example, when D exits, the P pointer in the most recent stack marker, the value 1485, will be loaded into the P register. That will cause execution to resume at C[1485], in the code segment—which is the location immediately following the instruction that called D.

**Figure 6-31. Nested Calls Create a Chain of Markers on the Stack**

## Returning a Value to the Caller

A function procedure can return a value back to its caller using the top of the register stack. This, like parameter passing, requires coordination between the caller and the called procedure. That is, the calling procedure must know the number of words constituting the return value.

This topic and the following one describe an example of a procedure, named “f”, that returns a value, and the instructions used to do so. This topic describes the procedure that will be called, as illustrated in [Figure 6-32](#). The following topic describes the calling of this procedure and how the caller retrieves the returned value.

The procedure for this example returns the square of a number, “x”. The procedure is written in TAL as follows:

```
INT PROC f (x);
  INT x;

  BEGIN
    RETURN x * x;
  END;
```

The instructions that perform this procedure are:

```
LOAD L -003      ! parameter x is obtained from L-003
LOAD L -003      ! load another copy of x
IMPY             ! squared result now exists in R[0]
EXIT 4           ! delete stack marker and parameter x
```

The procedure begins executing with RP=7 (empty). The first LOAD instruction loads the parameter “x” onto the top of the register stack. The procedure assumes that the caller has placed the parameter in location L[−3] (that is, immediately preceding the stack marker). The figure assumes that the passed value is 5.

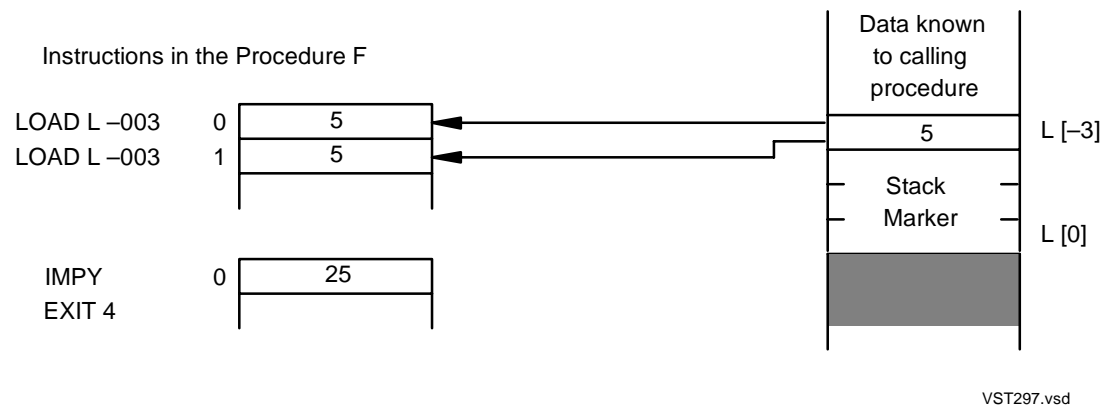
Following this first LOAD, the RP setting is 0. The second LOAD again loads the parameter “x”, because this is a procedure to square a number. Following this second load, the RP setting is 1.

The IMPY instruction multiplies the values in the register stack. The instruction eliminates both operands and leaves the result of the multiplication in R[0]. Following this operation, the RP setting is 0.

The EXIT instruction causes a return to the caller, deleting the parameter and stack marker (1 + 3 = 4) from the data stack. All that is left is the squared value on the top of the register stack, with RP=0.

Now proceed to the next topic for a complete sequence of passing a parameter, calling this procedure, retrieving the returned value, and using it in the calling procedure.

**Figure 6-32. Example of Returning a Value to the Caller of a Procedure**



## Retrieving a Returned Value

The preceding topic described a function procedure that is capable of returning a value to a caller. This topic describes the overall sequence of using that procedure in the context of a call and return. Refer to [Figure 6-33](#).

Suppose a call is made to the procedure “f”, as follows:

```
z := i + j - f(5);
```

That is, subtract the square of 5 from the sum of the contents of the variables “i” and “j”, then store the result in the variable “z”. The name “f” here means that the procedure “f” should be called, and “5” is the parameter passed to that procedure. Variables “i”, “j”, and “z” are local variables at L[1], L[2], and L[3], respectively. The instructions to perform this operation are:

```
LOAD  L +001    ! load i
LOAD  L +002    ! load j
IADD                     ! i + j
LDI    +005     ! load parameter to f
PUSH   711     ! push sum and parameter onto memory stack
PCAL   f       ! procedure call to f
STAR   1       ! move returned value from R[0] to R[1]
POP    100     ! bring saved sum back to R[0]
ISUB                     ! subtract returned value from i+j sum
STOR   L +003  ! store result into z
```

The first three instructions calculate the sum “i” + “j” and leave the result in R[0]. The LDI +005 instruction loads the parameter to “f” to the top of the register stack at R[1].

The PUSH instruction pushes R[0:1] onto the memory stack. Following the PUSH, the two top-of-memory-stack locations contain:

```
S[-1] = sum of i + j
S[0]  = 5, the parameter to f
```

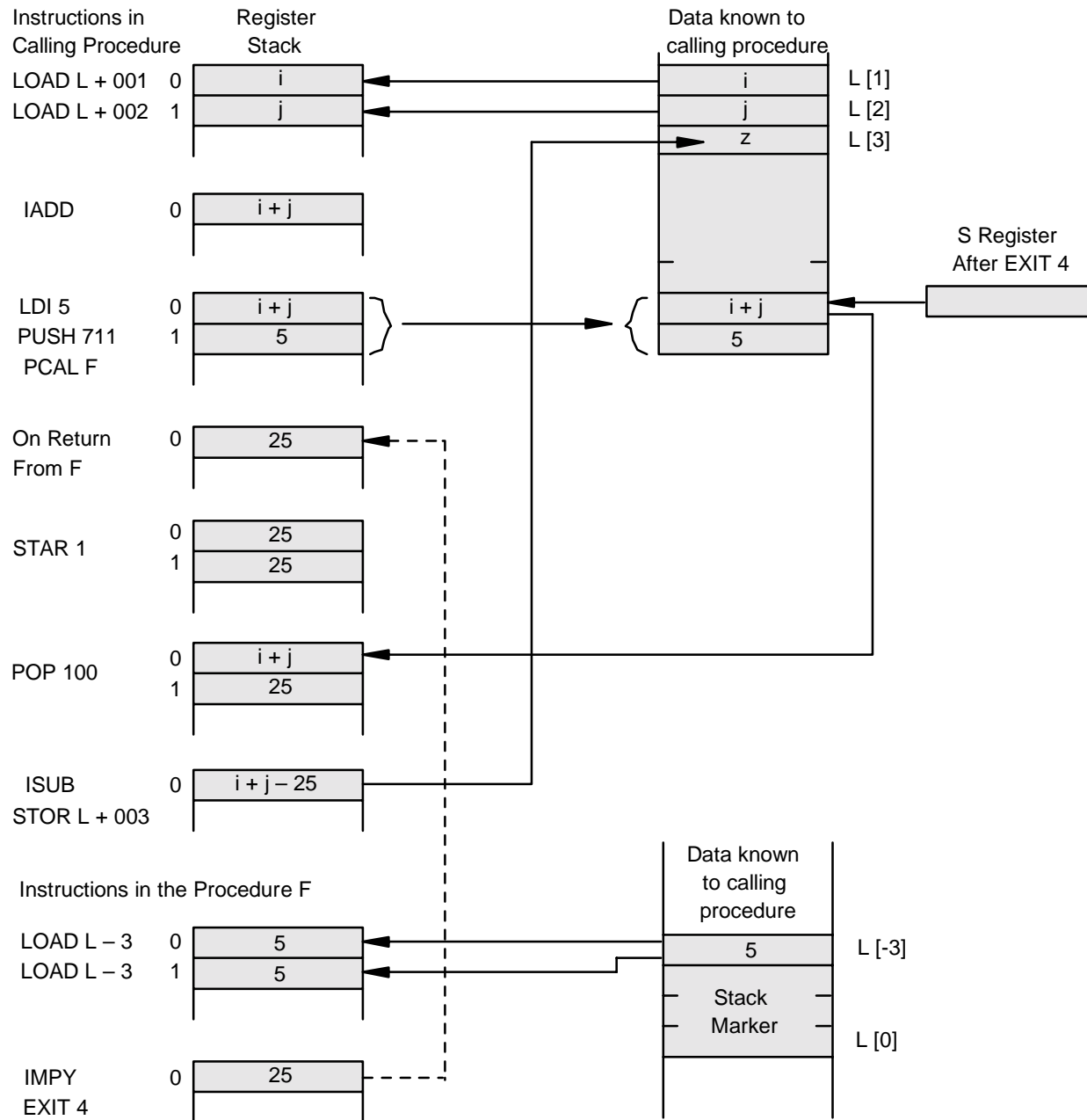
This clears the register stack for use by the procedure, which now is invoked by the PCAL instruction. (The PCAL instruction specifies the PEP number of the “f” procedure.) On the return from “f”, R[0] of the register stack contains the square of 5.

The STAR instruction moves the return value in the R[0] register stack location to R[1] in preparation for the subtraction from the sum of “i” + “j”.

The POP 100 instruction brings the sum of “i” + “j” (calculated previously) into R[0] and sets RP to 1 (to point to the returned value).

The ISUB instruction subtracts the return value of “f” from the sum of “i” + “j”. The STOR instruction stores the result in the variable “z”, which is the location L[3], and RP becomes 7 (empty).



**Figure 6-33. Example of Retrieving a Value Returned by a Called Procedure**

VST298.vsd

# Subprocedure Calls

A procedure itself can contain one or more **subprocedures**. The following table summarizes the differences and similarities between procedures and subprocedures.

<b>Procedure</b>	<b>Subprocedure</b>
Can have parameters	Can have parameters
Can access local and global storage	Can access local, sublocal, and global storage
Temporary storage area: 160 TNS words	Temporary storage area: 32 TNS words
Local addressing modes: L-plus, L-minus	Sublocal addressing mode: S-minus
Uses PCAL, XCAL, DPCL, EXIT instructions	Uses BSUB, RSUB instructions
Has its own callability attribute	Always defaults to the privilege of the containing procedure
Can be called from other procedures and their subprocedures	Can only be called from the local procedure and its subprocedures
Can call subprocedures within itself (only)	Can call other subprocedures (same procedure)

Subprocedures are invoked using the BSUB (branch to subprocedure) instruction. Because BSUB is a branching instruction, the subprocedure entry point is calculated as a self-relative address. Execution of the BSUB instruction differs from other branching instructions in that it places a return address on the top of the memory stack. This is shown in [Figure 6-34](#). (The right-hand part of this figure shows the same data area three times.) Note that before BSUB executes, the subprocedure parameters must be pushed onto the stack. Then BSUB does the following:

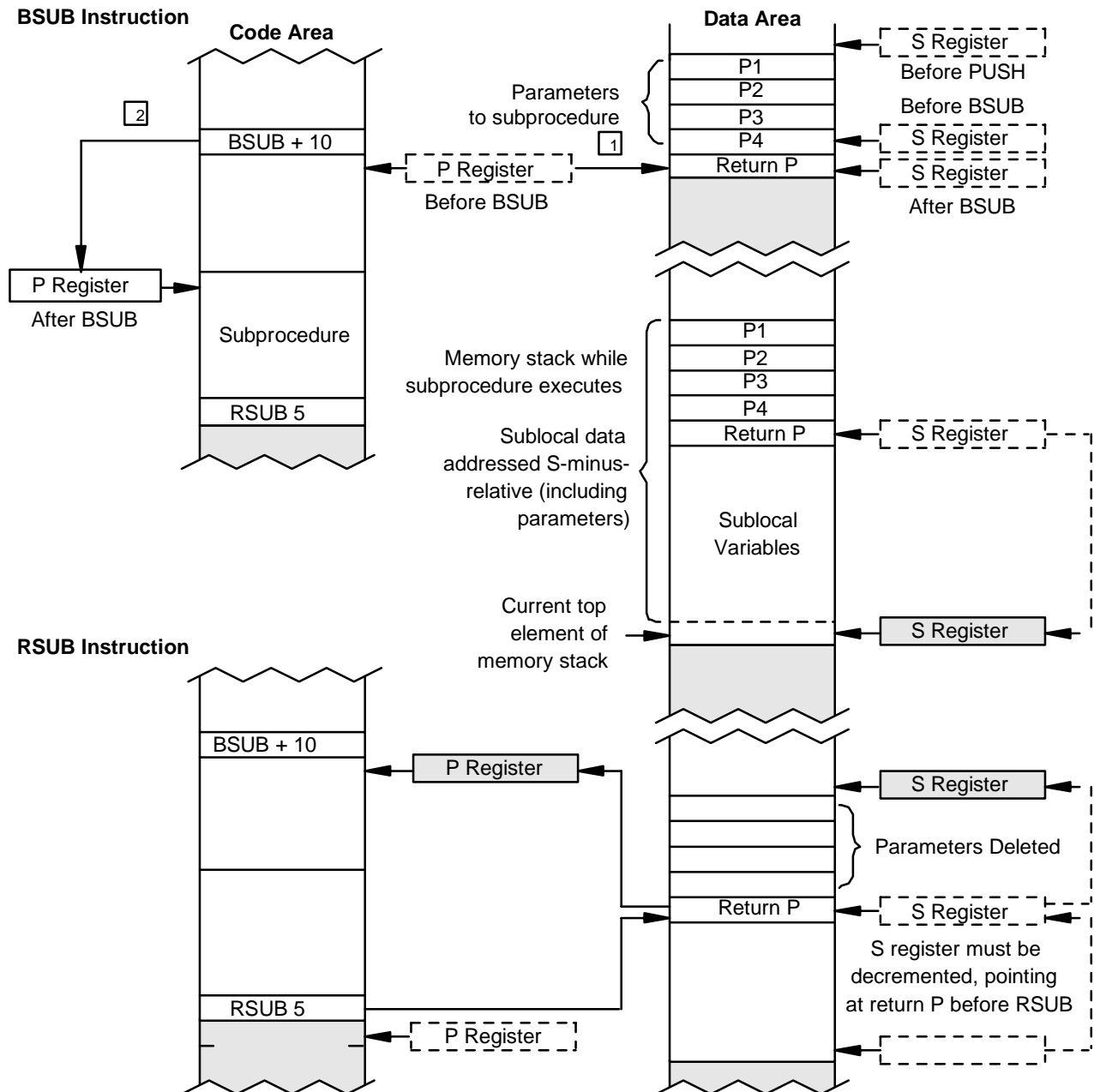
1. Places the return address (that of the instruction following BSUB) on the stack.
2. The branch address of the subprocedure is put into the P register.

The subprocedure uses variables and parameters in the sublocal area. This area is addressed using the S-minus addressing mode, providing direct access to the 32 locations below and including the current S register setting—that is, S[–31:0].

The last instruction that a subprocedure executes is an RSUB (return from subprocedure) instruction. Before RSUB executes, the S register must be decremented, pointing at the return P location. The RSUB instruction returns control to the instruction following the BSUB instruction by putting the return address, at the current top of memory stack location, into the P register:

```
P := data [S];
S := S - S^decrement;
```

The “S^decrement” value (which is specified in RSUB) is used to move the S register setting below the sublocal data area, thus deleting the one-word return address and all parameters that were passed.

**Figure 6-34. Subprocedures Return Values Through Sublocal Data Area**

VST299.vsd

# Calling External Procedures

Procedures in another code segment can be called almost as efficiently as the current segment's own procedures. Two important features that make this possible are the **space ID** (identification) convention and the XCAL (External Procedure Call) instruction. The space ID convention provides a simple, one-step means to return to the segment of the caller (only the Environment register needs to be examined), and the XCAL instruction provides a two-table look-up sequence to get to the exact entry point of the external procedure.

As is usual when any procedure is called, the first action of the calling instruction (XCAL in this case) is to save the calling environment in a stack marker. The stored copy of the Environment register contains the complete segment identification of the caller's segment. For procedure calls that are internal within a single code segment (as performed by the PCAL instruction previously described), the segment identification is not needed. However, the EXIT instruction always checks this information so that it can return in exactly the same way regardless of which instruction made the call.

For external calls, though, the complete segment identification is very important because the call can be made to any one of four code spaces (user code, user library, system code, or system library) and each of those code spaces can have multiple segments. Thus, on return, the environment must switch quickly back to the specific space and specific segment within that space.

The complete segment identification is in the saved copy of the Environment register, shown in [Figure 6-35](#). The complete identification consists of the LS and CS bits, to select one of the four code spaces, plus a space ID index to select a specific segment within that code space. The space ID index occupies bits 11 through 15. Normally, in the Environment register itself, these bits are used to contain the condition code and register pointer (CC and RP). However, because CC and RP do not need to be saved, these bits are available to save the space ID index in the stack marker copy.

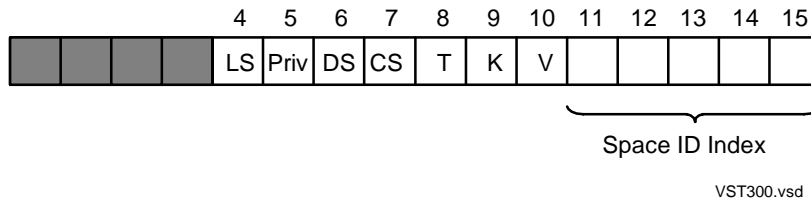
In the saved copy of the Environment register, in the stack marker, bits 4 through 10 contain exactly the same single-bit fields that exist in the Environment register itself. LS, DS, and CS identify, respectively, library space (0 = no, 1 = yes), data space (0 = process, 1 = interrupt), and code space (0 = user, 1 = system). PRIV, T, K, and V indicate (if set = 1), privileged mode, traps enabled, carry, and overflow. Bits 11 through 15, the space ID index, identify one of 32 possible TNS code segments in the space specified by the combination of LS and CS.

After the XCAL instruction has placed the three-word stack marker on the top of the user stack, it then moves L and S in the same manner as a PCAL instruction (that is, defines a new stack frame). However, instead of transferring control directly to a procedure within the segment, control is transferred out of the segment. This transfer of control is accomplished through two tables. The first, in the caller's segment, is called the **external entry point table**, or XEP table. The second, in the segment of the called procedure, is that segment's PEP, or procedure entry point table (previously described for PCAL).

The next topic illustrates an example of the operations involved in an external procedure call.

---

**Figure 6-35. Saved Copy of the Environment Register Preserves Space ID Index**



## Example of an External Procedure Call

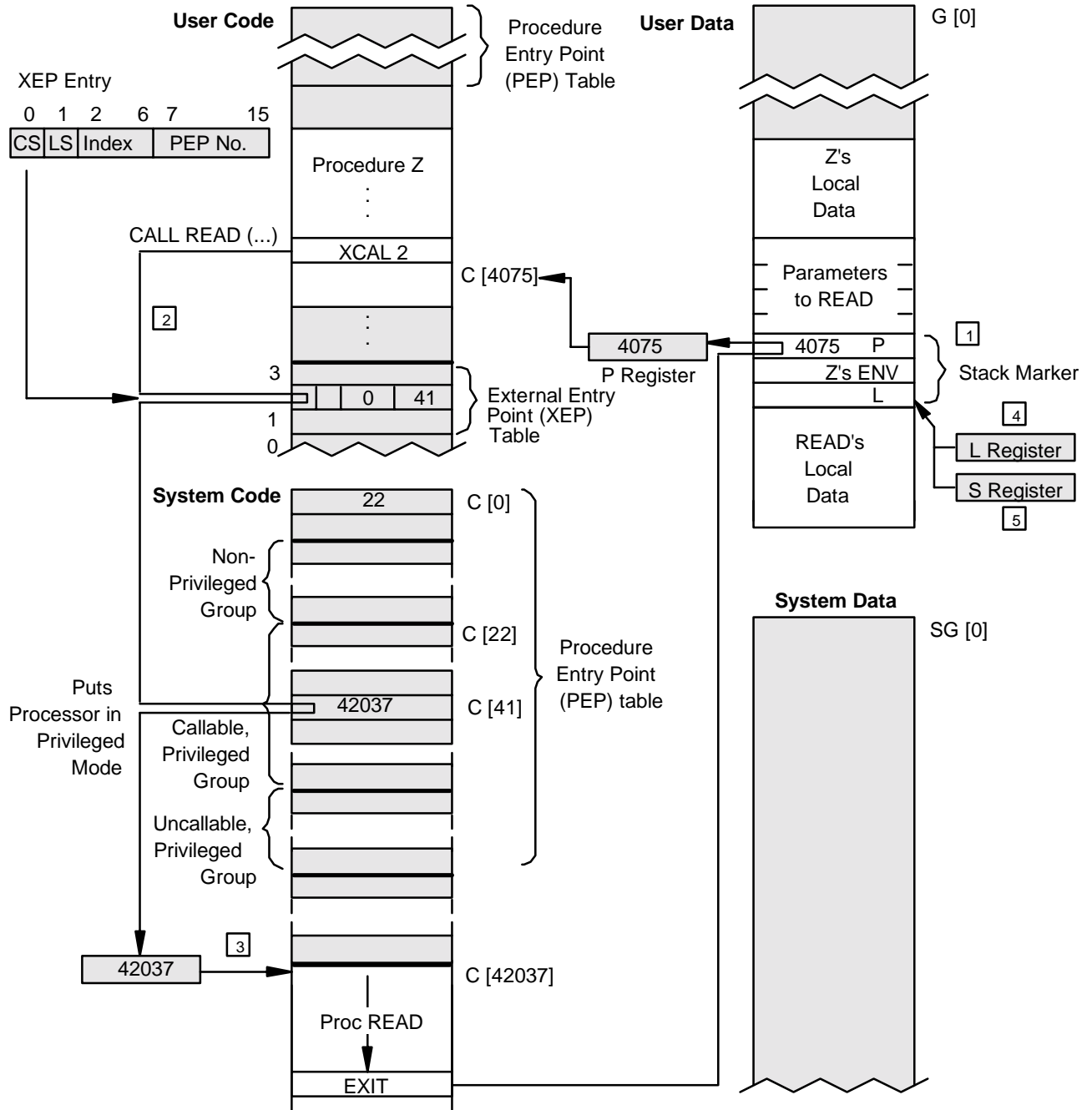
In this example, procedure Z in user code calls a READ procedure in system code. The XCAL instruction that implements this call refers to an external entry point table in its own segment and a procedure entry point table in the segment that contains the called procedure. These references provide the entry point address.

The general flow applies to any allowable external call between any pair of segments in any of the four code spaces—user code, user library, system code, and system library.

With reference to the numbered callouts in [Figure 6-36](#), the sequence of events is as follows:

1. The caller's environment is stored in a stack marker. The stored copy of the ENV register contains the complete segment identification (LS and CS bits, plus the space ID index) of the caller's TNS code segment.
2. The C[0]-relative address of the procedure being called is obtained by a three-step process. First, the XCAL instruction locates entry number 2 in the caller's external entry point (XEP) table. Second, that XEP table entry (formatted as shown upper left) is used to locate the desired code segment (CS and LS specify the system code space, in this example, and bits 2 through 6 specify a space ID index of 0) and a particular procedure entry point number (41) as an index into the procedure entry point (PEP) table. Third, the address in that PEP table entry (42037) is put in the P register so that the next instruction executed will be the first instruction of the system procedure.
3. If the calling procedure is not executing in privileged mode (assume it is not), the callability attribute of the system procedure being called is checked. In this case, the PEP number of 41 puts the call into the callable, privileged group. The call is therefore permitted; the procedure will execute in privileged mode but will restore the mode to nonprivileged on exit.
4. The S and L registers are set with the G-relative address of the new top-of-stack location. The new L register setting defines the base of the stack frame for the system procedure being called.
5. The new S register setting is tested for an address within the memory stack area, G[0:32767]. If the value is greater than 32,767, control is transferred to the stack overflow trap and the XCAL instruction is aborted.
6. The CS bit of the Environment register (not shown) is set to 1 and the LS bit is set to 0 to indicate that further code area references will be in the system code space (segment 0 in this example). Also, the register stack pointer, RP, is given an initial value of 7 (stack empty).

When the system procedure finishes, the EXIT instruction is executed. The CS and LS bits, plus the space ID index bits from the stored copy of the ENV register, are used to reestablish the caller's segment of the user code space as the currently selected code space, so that the next instruction is executed from that segment.

**Figure 6-36. Sequence of Events for an External Procedure Call**

VST301.vsd

# Resolving Virtual Addresses for External Calls

The preceding few topics assume that a P-register address is all that is needed to identify the entry point of a procedure in memory. However, that is simply a word address that is relative to the start of some TNS code segment; it does not tell you where that address is in virtual memory. So, given a 16-bit word address for a P-register address, what is the corresponding address in virtual memory?

The illustration shown in [Figure 6-37](#), based on the example explained below, illustrates how the resolution to virtual address is made. The process code is assumed to be contained in the first four segments of the region. The segments use, respectively, four pages, three pages, five pages, and four pages. (The first page in all cases contains the procedure entry point table, PEP, and some of the code; the external entry point table, XEP, is on the last page, with entries assigned backward from the page boundary.) All addresses and offsets are hexadecimal numbers.

The first step is to identify the base address for the particular code space. As explained in [Section 4, Memory Addressing and Access](#), the four code spaces have fixed starting addresses in the process address space. Three of them are in separate regions of the nonprivileged space, and the system code (SC) is in the Kseg0 part of the privileged space.

The four base addresses are:

```
UC   %h 70000000
UL   %h 72000000
SL   %h 7A000000
SC   %h 80000000
```

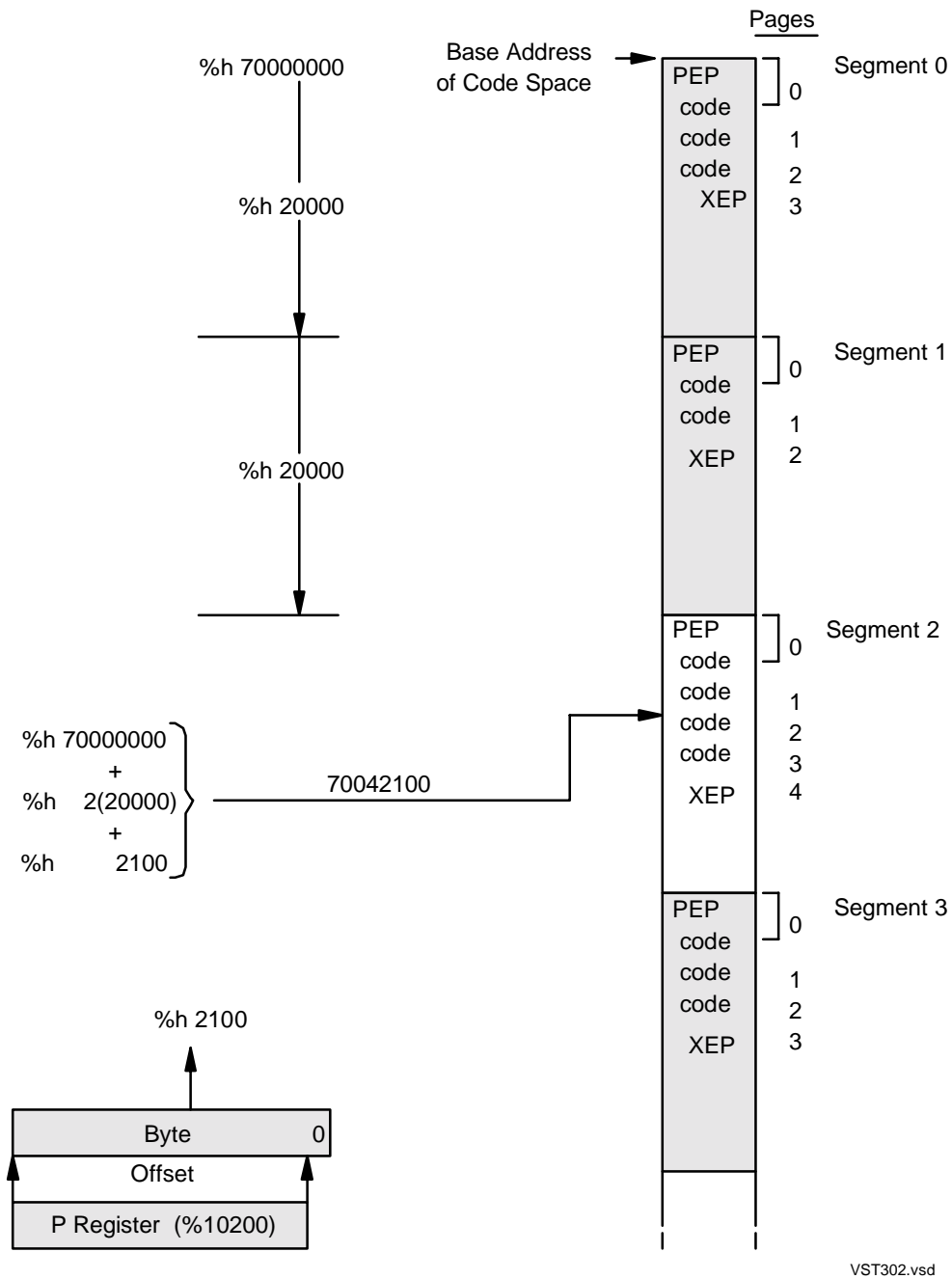
The example in the figure assumes that the user code is being addressed. Therefore, the base address is %h 70000000.

The next step is to compute the segment offset. This is done by multiplying the space ID (2, in this case, for segment 2) by the maximum possible segment size, which is 128 KB (20000 in hexadecimal) even for code segments that are not full. This gives an offset of %h 40000.

The next step is to get a hexadecimal byte address for the relative address in the P register. The byte address is obtained by shifting the word address one bit position. The figure assumes that P contains %h 1080 (= %10200); the shifted address is %h 2100—that is, %h 100 bytes into the third page. (A page is %h 1000 bytes.)

The final step is to add these three values together. In the example, the actual virtual address is %h 70042100.



**Figure 6-37. Example of Resolving a Virtual Address in User Code Space**

# An Accelerated Program File in Virtual Memory

An accelerated program file, when it is allocated space in absolute virtual memory as an executable process, occupies some number of consecutive absolute segments in the Kseg2 part of the privileged space. That is illustrated on the right side of [Figure 6-38](#). The illustrated example assumes that the original TNS code is contained in two absolute segments (A and A+1) and the accelerated code is contained in eight absolute segments (A+2 through A+9). Shading denotes unallocated space.

When that same program file becomes a code region in nonprivileged space, the relative segment number assignments are such that the TNS code always begins at relative segment 0 of the region, and the Accelerator-generated RISC code always begins at segment 32. That is illustrated on the left side of [Figure 6-38](#). The illustration expands the assigned segments to show details that will be described in the remaining topics of this section.

Note that the first couple of segments generated by the Accelerator (in this example) contain data, not code. This data, beginning at segment 32, consists of a small (2 kilobyte) header that describes the remainder of the area, and a set of tables called **Pmap tables**. The function of Pmap tables is to provide a cross-reference between locations in the TNS code and corresponding locations in the accelerated code. There is one Pmap table for each segment of original nonaccelerated code; in this case, there are two. (If the program file was not accelerated, this fact would be indicated in memory by the presence of a page of zeros in place of the header.)

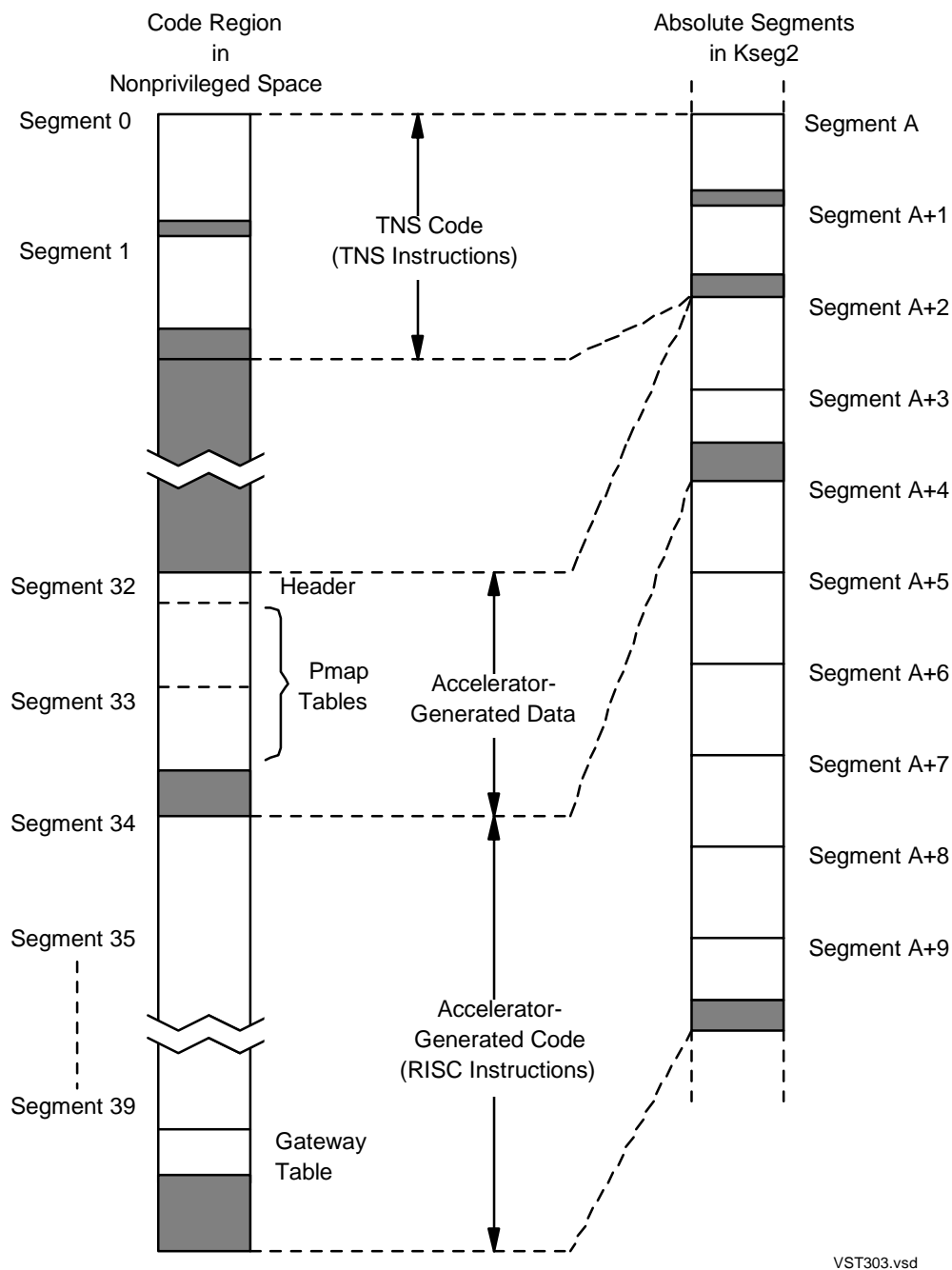
After the Pmap tables, beginning at the next segment boundary (in this case, segment 34), is the accelerated object code. The accelerated object code consists of directly executable RISC instructions. The illustrated example assumes that this code occupies the six relative segments numbered 34 through 39.

As indicated, the RISC code itself recognizes no code-segment boundaries; the accelerated object code is a single, integral entity. Thus there is no need for “external entry point” tables, as in the TNS environment. Execution can pass from any location to another with a single, direct jump anywhere within this area, even up to the maximum achievable size. (This capability also contributes to the “acceleration” of the code.)

At the end of the accelerated object code may be another table, called the **gateway table**, which is present if the accelerated object code contains any procedure with the CALLABLE attribute.

If there is any resident code in the program, the gateway table and the header are also made resident. In addition, if any part of the TNS code is resident, the corresponding parts of the accelerated code and Pmap table are also made resident.

**Figure 6-38. Accelerated Programs Files Contain Both TNS and Accelerated Code**



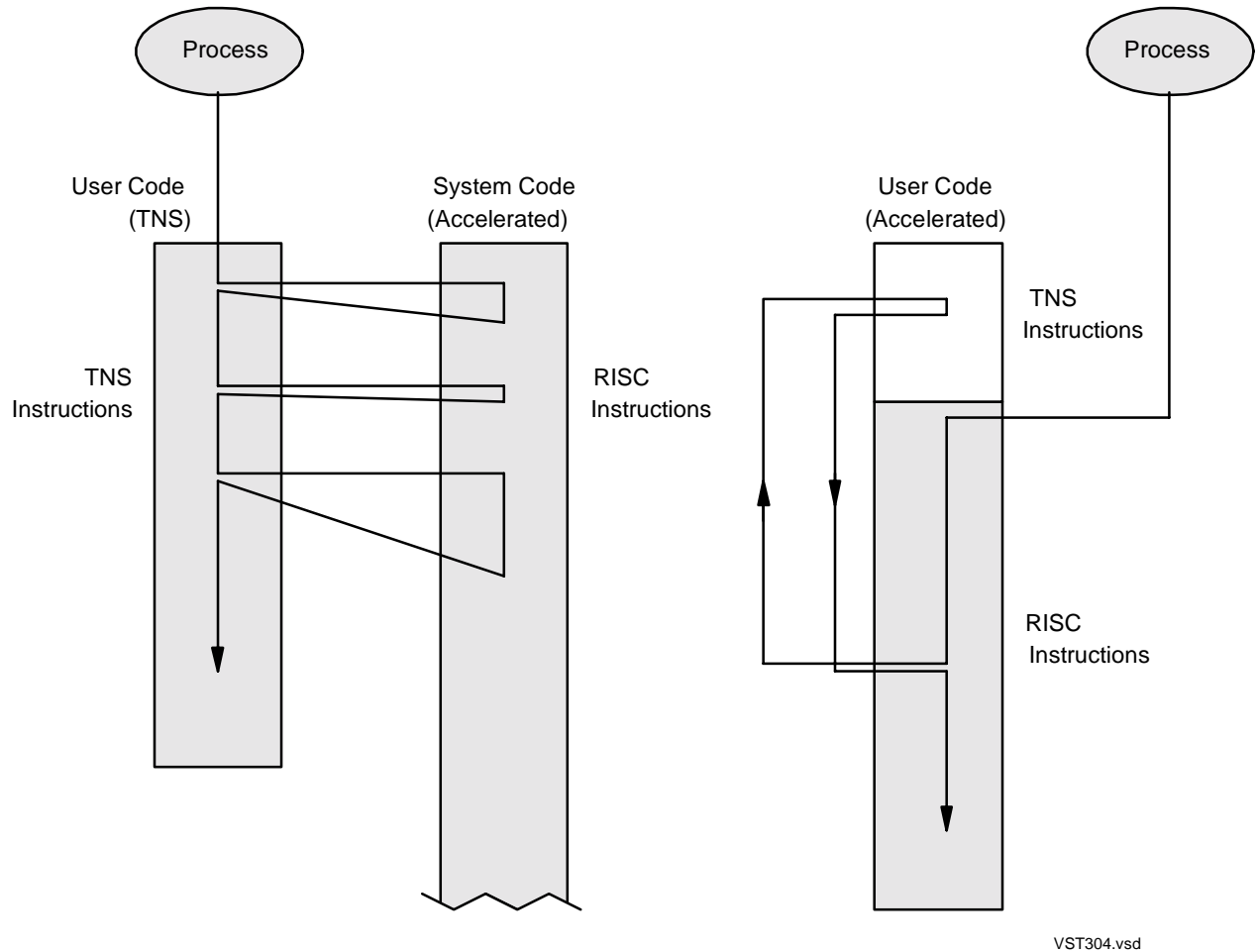
# Execution Mode Switches

Although one might expect a TNS code file to execute exclusively in the TNS execution mode and an accelerated code file to execute exclusively in the accelerated execution mode, there are occasions for temporary switches to the opposite mode. Two of the main reasons for such switches are illustrated in [Figure 6-39](#).

The first case, shown on the left, involves a process that uses a code file that was never accelerated. This process begins its execution in the only form of code that it has—TNS object code. However, each time it makes a call to any code that has been accelerated, such as to a procedure in the system library, there is a switch to the accelerated execution mode. This occurs because the millicode always selects the accelerated form of code if it is present. If the program makes many calls to system code and system library, there will be many switches to accelerated mode.

The second case, shown on the right, involves a process that uses a code file that was accelerated. Accordingly, it begins its execution in the accelerated object code portion of its object file. It may continue to the end without ever switching to TNS mode. However, if the Accelerator program, when it was executed, was unable to translate a particular section of TNS instructions to accelerated code, then it is necessary to return temporarily to the original TNS instructions to determine the proper way to continue.

In the first case, the interpreter millicode takes care of both the call and the return from the system procedure. As explained previously, it is the millicode that is executing and not the TNS instructions. In the second case, the Accelerator will have encoded a jump to millicode at the point where it was unable to translate a section of TNS code, along with the appropriate TNS instruction address. The return to accelerated code usually occurs at the next transfer of control point, such as at a procedure return or call (but not at ordinary branches).

**Figure 6-39. Switching Modes for System Calls and Translation Assistance**

# Procedure Return in Accelerated Code

Besides actual mode switches, as described in the preceding topic, there are other times when an accelerated program needs access to TNS information. One important example is the return from any procedure call. In this case, what is needed is the address derived by the EXIT instruction from the stack marker, rather than the next RISC instruction.

Because the memory stack has exactly the same layout and content as on TNS processors, there is no room in the stack marker for a 32-bit RISC return address. Accelerated EXIT and RSUB instructions therefore work only from 16-bit TNS return addresses.

The basic TNS mechanism of a procedure call and return is shown in the left part of [Figure 6-40](#). A call instruction (PCAL in this example) saves its return address (a TNS P value) in a stack marker and jumps to the entry point of the procedure (N in this example). After the called procedure has run to completion, it executes an EXIT instruction. The EXIT instruction uses the saved TNS P and ENV values in the stack marker to return to the instruction following the calling instruction.

The RISC instruction set does not include equivalent instructions to perform these complex operations. Thus the simple jump instructions must be supplemented by a separate mechanism that accomplishes the functions of the TNS EXIT instruction. The complete call and return sequence is shown in the right part of [Figure 6-40](#), in five steps. This is the RISC equivalent (accelerated code) of the TNS code shown to the left.

The accelerated code executes a jump to the entry point of the called procedure (Step 1). Then the stack marker is built on the TNS data stack. (In contrast to the TNS mode sequence, the stack marker is built after jumping to the entry point.) At the conclusion of the called procedure, the code executes a jump to the EXIT millicode routine (Step 2). The EXIT millicode reads the TNS P value that was saved in the stack marker (Step 3). This P value is the C-relative address of the next TNS instruction to be executed following the PCAL. The saved Environment register (also in the stack marker) provides the correct segment identification (space ID).

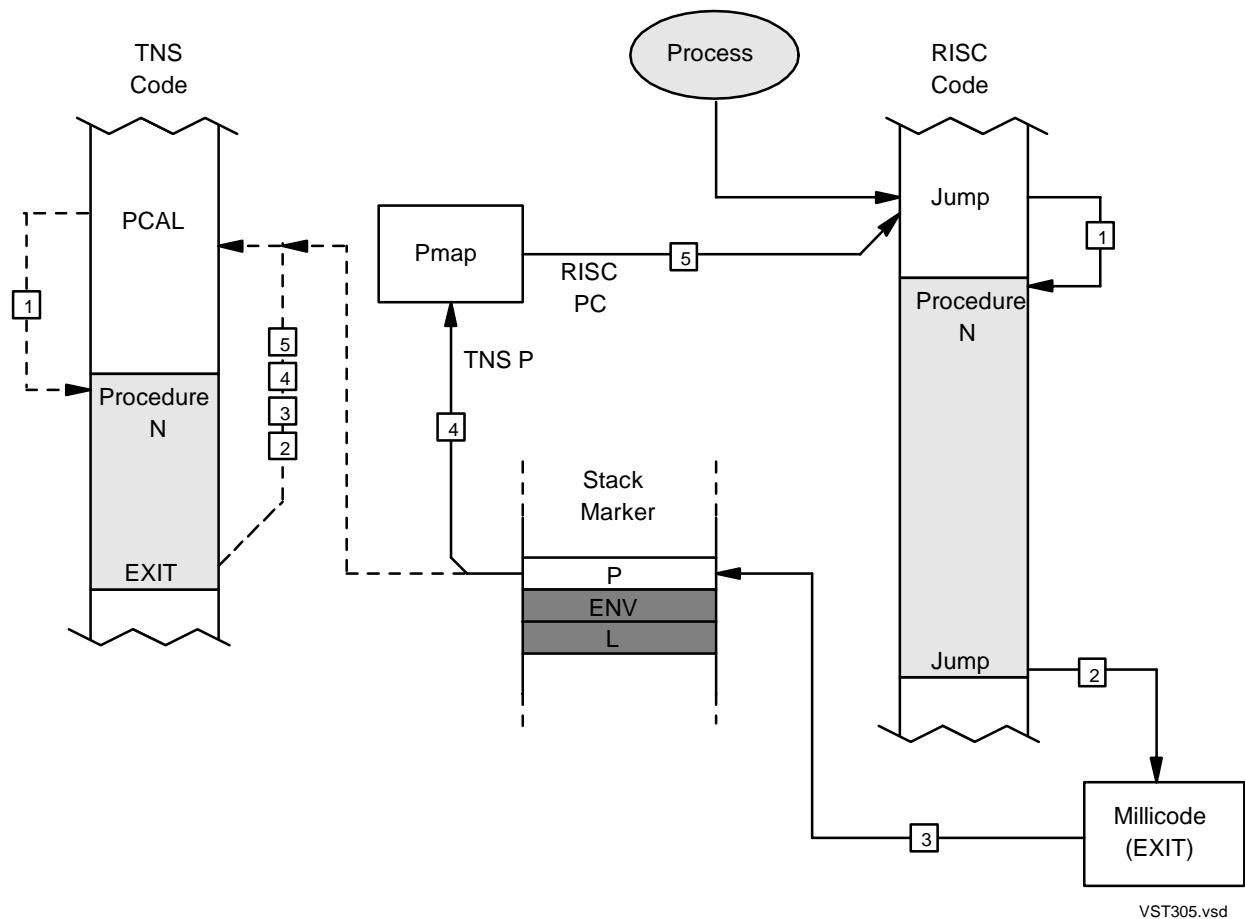
Now the EXIT millicode refers to the Pmap table for this TNS code segment and finds the starting point of the RISC instructions for that next TNS instruction (Step 4). After forming a complete address out of the information in the Pmap table, the EXIT millicode performs a jump to the appropriate point in the RISC code (Step 5).

The Pmap table may indicate that the TNS return address is not a “register-exact point” (see Pmap discussion in the next topic). In that case, the EXIT millicode jumps to the TNS interpreter millicode to execute the next portion of the program in nonaccelerated mode.

In summary, with reference to [Figure 6-40](#): (1) The jump to procedure N in RISC code is equivalent to the jump performed by PCAL. (2) The final action of procedure N is to execute a jump to EXIT millicode. (3) The EXIT millicode fetches the TNS P value of

the return point from the stack marker. (4) Using that P value, the corresponding RISC address is found in the Pmap table. (5) EXIT performs a jump to that address.

**Figure 6-40. Accelerated Procedure Return Requires Access to TNS Information**



# Mapping Return Addresses and Debug Points

The purpose of the Pmap table is to map TNS P addresses in a given TNS code segment of an accelerated code file to corresponding RISC PC addresses. In theory, that could result in a large table (64K words, or 256 KB, for each TNS code segment). In addition, two tables are really needed: one to map return addresses for procedure returns and one to map debug points.

This topic explains how an efficient encoding of the Pmap table allows two logical tables to be combined into one and overall size to be only 96 KB. However, the details are not essential to comprehension of the overall function and may safely be skipped.

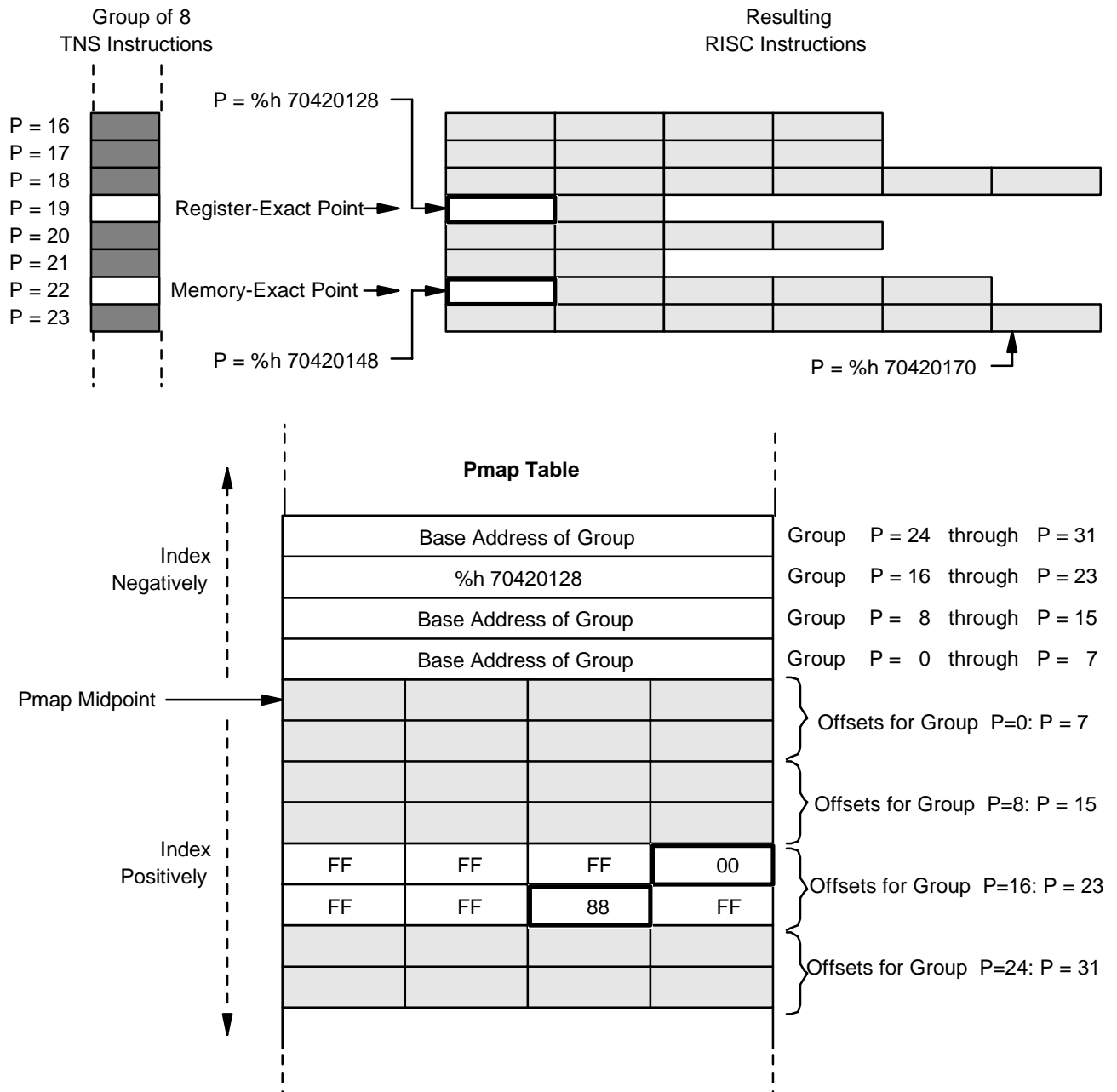
The part of the table that maps return addresses is called the Rmap. The **Rmap** maps a “safe” TNS address to its corresponding RISC address. Safe points are called **register-exact points**. These are places in the accelerated program at which control can safely transfer from nonaccelerated to accelerated execution mode (or during returns, can stay in accelerated mode). The part of the table that maps debug addresses is called the **Dmap**. The Dmap maps a TNS **memory-exact point** (usually a statement boundary) to its corresponding RISC address. Most register-exact points are also memory-exact points; thus a single table for both maps is practical.

For compactness of the table, full 32-bit addresses are actually entered only for the start (or base) of groups of eight TNS addresses. The drawing in the top left area of [Figure 6-41](#) shows an example of such a group. Because the range of addresses shown is 16 through 23, this would be the third group in the table. (Though called P values, these are C-relative addresses.) Assume for this example that P = 19 is a register-exact point and P = 22 is a memory-exact point.

The diagram at top right assumes that 33 RISC instructions resulted from accelerating the TNS code. (The correspondence of TNS-to-RISC instructions is simplified.) The base for this group is the first usable map point (for P = 19), which is assumed to be RISC address %h 70420128. The next map point (for P = 22) is offset eight instructions from this base. The Pmap table provides RISC addresses for both points.

The Pmap table, as shown, consists of two arrays back-to-back, with the midpoint taken as the base of both. The base address is found by indexing negatively (divide 19 or 22 by 8, to arrive at the third entry). Offset values are found in groups of eight specially encoded individual bytes, and the specific group is found by indexing forward from the midpoint (note highlighted group for the example). The most significant bit of each byte specifies whether the entry is an Rmap entry (0) or a Dmap entry (1), with certain values having reserved meanings (all ones, or %h FF, means no entry). The remaining seven bits specify the offset—in this case %h 00 and 08 (ignoring the high-order bit). Shifted by two to produce a byte address, these offsets become 00 and 20, respectively. When added to the base, the addresses are %h 70420128 and 70420148.



**Figure 6-41. Pmap Is a Cross-Reference of Code Segment Addresses**

VST306.vsd

# Gateway Tables

When a process that is operating in TNS mode calls a CALLABLE system procedure, there is necessarily a transition to privileged mode.

In the case of programs that use TNS mode, such a transition to privileged mode takes place in the millicode for the calling instruction (XCAL, PCAL, or DPCL). That is the normal case for the TNS architecture, and it is based on the layout of the procedure entry point (PEP) table for the segment containing the called procedure. However, in the case of accelerated programs, use of the PEP table is generally skipped. Thus there needs to be a separate mechanism for providing secure transitions to privileged mode, and that is the function of the **gateway tables**.

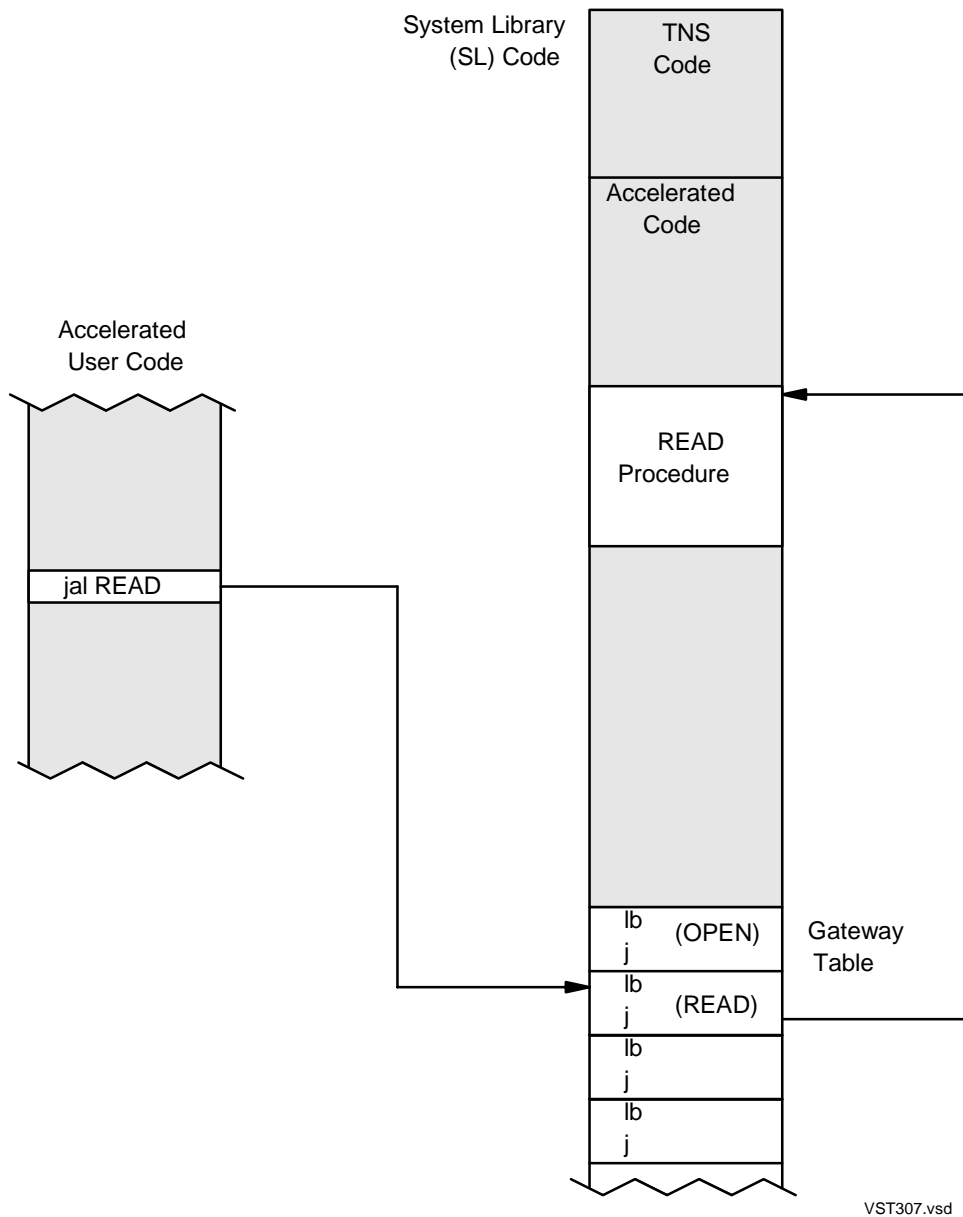
Gateway tables are built by the Accelerator and, as shown in [Figure 6-42](#), immediately follow the accelerated code in the object code file. Gateway tables exist only for those code areas that actually contain callable privileged procedures. User code (UC) and user library (UL) seldom have such CALLABLE procedures and so rarely have gateway tables. In fact, most CALLABLE procedures are in the system library (SL) area. The example shown in [Figure 6-42](#) illustrates the gateway table for the system library, which resides in user space. (Calls to callable procedures in the SC area require an additional mechanism to make the “far jump” to SC; for efficiency, the far-jump mechanism is combined with the gateway table and so calls to SC are described separately in the next topic.)

Note the major elements of the example shown, all highlighted in white. These are the calling instructions in the accelerated user code (on the left) and the gateway table and called procedure code in the accelerated code part of the system library code (on the right). The example assumes the procedure being called is the READ procedure.

As indicated, the last instruction of the calling instructions in the user code is a jump to the appropriate entry (READ) in the gateway table. Each entry of the gateway table consists of two instructions: a special load-byte instruction and a jump instruction. The load-byte instruction attempts to load a certain byte of the scratchpad page (SPAD), which is located in privileged memory (Kseg2). Such a reference requires privileged mode. For the privileged caller, the load occurs and the jump immediately follows. But for the nonprivileged caller, an address error exception occurs. This invokes the exception handler, which checks to assure that the exception occurred within the gateway table, and allows the process to proceed in privileged state.

The exception handler returns to the first instruction of the gateway; this time the load-byte instruction completes without incident, and the jump instruction sends control to the entry point of the called procedure (READ in this example). The called procedure now executes in privileged mode and performs the indicated operation. The EXIT millicode, at procedure's end, sets the RISC status register back to the state of the caller (nonprivileged in this case) and exits back to the user code.

**Figure 6-42. Gateway Tables Provide Privileged-Mode Transitions for Accelerated Code**



# Far Jump Tables

In computing a 32-bit PC address from the 26-bit target field of a RISC jump instruction, the RISC processor takes the four high-order bits from the current program counter value. That fact results in 16 possible **direct jump areas** in the 4-GB virtual address space, each being 256 megabytes. Most process code (UC, UL, and SL), as well as the interpreter and nonprivileged instruction set millicode, are in the last direct jump area of user space. System code (SC) and some privileged millicode, however, are in the first direct jump area of kernel space. Jumps between these two areas are accomplished by means of far jumps.

A **far jump** is an entry in a **far jump table** consisting of four or more RISC instructions. These instructions use a full 32-bit target address to enable crossing the boundaries of direct jump areas.

There are only two far jump tables for accelerated code, both built by SYSGENR. One is at the end of the accelerated code of SL and is combined with SL's gateway table, and the other is at the end of SC. (SC has no need of a gateway table.) All calls to SC (or Kseg0 millicode), whether they originate in UC, UL, or SL, must be routed through the single far jump table in SL. Any calls in the reverse direction, from SC, would only be to SL (or the millicode in user space).

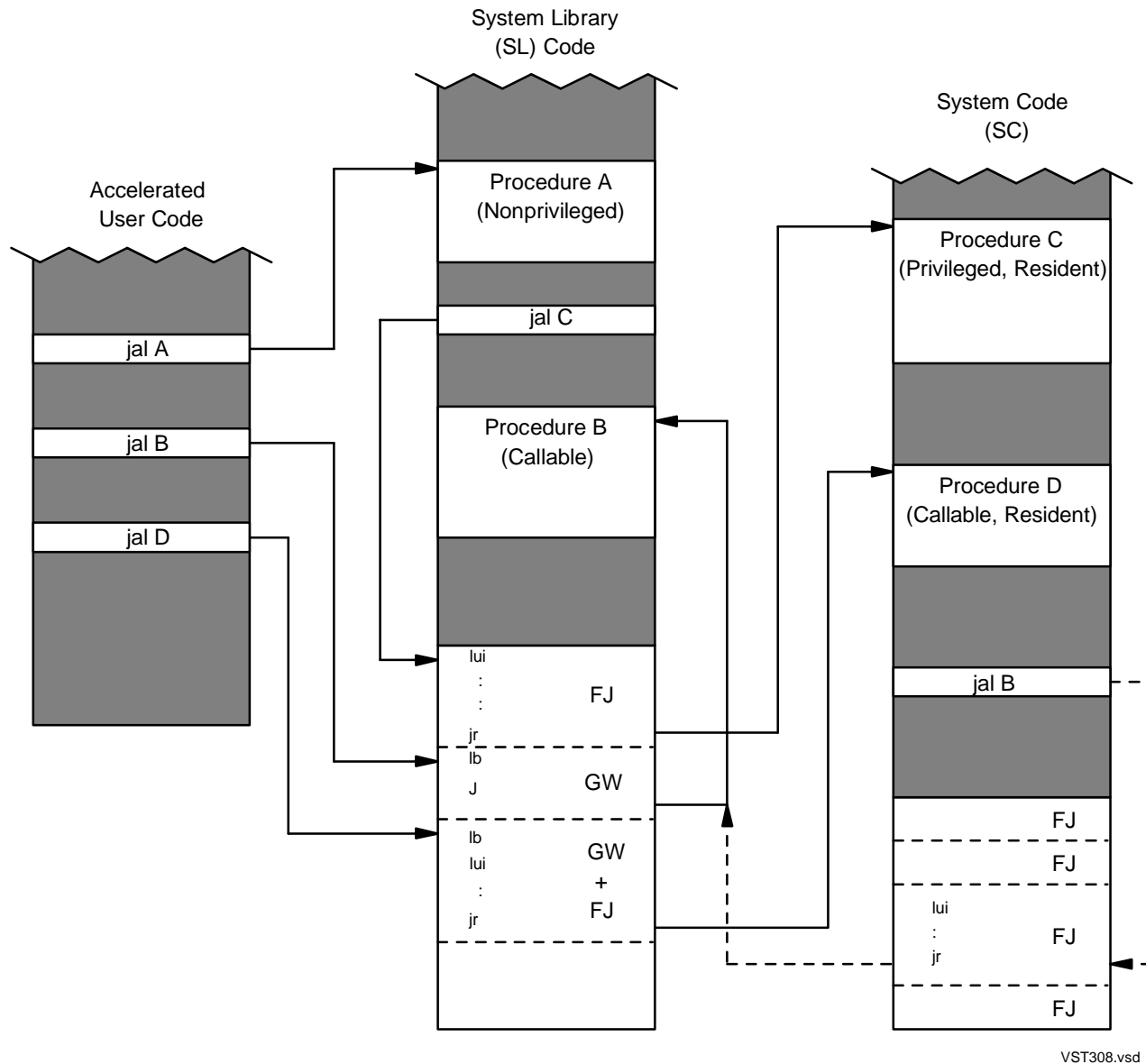
[Figure 6-43](#) illustrates five common cases of calls between the three main kinds of code areas: user code (UC or UL), SL, and SC. Three of the calls originate in user code (calls to A, B, and D), one in SL (call to C), and one originates in system code (another call to B). The labels FJ, GW, and GW + FJ designate whether the entry is a far jump table entry, a gateway table entry, or a combined entry, respectively.

Procedure A is a nonprivileged procedure located in SL. Because it requires no privileged mode transition, it requires no gateway table entry, and because it is in the same direct jump area as the example call, it requires no far jump table entry.

Procedure B is a callable procedure in SL. Because this procedure is callable, a gateway table entry is required. When the procedure is called from user code, the call must jump first to the gateway table entry. As described in the preceding topic, this indirect access makes the privileged-mode transition before jumping to the procedure. However, when B is called from SC (dashed lines), the first jump is to the far jump table entry in SC, which contains a jump-register instruction that jumps directly to B, ignoring the gateway table because any code that runs in SC is already privileged.

Procedure C is a privileged (but not callable) procedure located in SC. It can be called only by privileged code and therefore requires no gateway table entry. When it is called from SL, the call jumps first to the far jump table entry for procedure C; the final instruction of this entry completes the jump to procedure C code in SC.

Procedure D is both callable (requiring a gateway table entry) and located in SC (requiring a far jump table entry). Both entries are combined, including the instructions needed for both functions, and including the jump instruction that completes the jump to procedure D.

**Figure 6-43. Far Jump Tables Are Needed for Calls To and From System Code**

# Maintaining TNS State Values

The 32 general-purpose registers used by the RISC processor are numbered \$0 through \$31. \$0 is hardwired to contain the value 0 and so is not considered in usage assignments. Many of the registers are used by millicode or accelerated code for expression evaluation, parameter passing, and so on. Those that are used to maintain TNS state information fall in the range \$14 through \$30, as shown in [Figure 6-44](#).

Some registers are assigned differently depending on whether the current mode of execution is to execute the interpreter millicode or to directly execute accelerated object code. Registers \$16 through \$23 contain the TNS register stack values while in accelerated mode but not in nonaccelerated mode. (In the latter case, R0 through R7 are maintained in memory, in the RP wrap page, as previously described.) The other seven registers, which are assigned to maintaining TNS state (unshaded), are assigned consistently in both modes.

The carry and condition code values, although normally accessed as fields of the ENV register in the TNS environment, are each assigned to whole registers in TNS/R processors, because of frequent updates. The Carry (K) register can have one of two values: 0 (no carry) or 1 (carry). The Condition Code can have three kinds of values: 0 (equivalent to N=0 and Z=1), any positive value (equivalent to N=0 and Z=0), or any negative value (equivalent to N=1 and Z=0).

In both modes, registers \$24 and \$25 contain the user code address and ENV register fields. “User code segment” is a pointer to the base of the most recently used TNS code segment in the user code (UC) region (relative segment 3). “ENV” reflects the maintained state of the TNS Environment register in stack marker format, with bit assignments as follows: instruction breakpoint bit (1), LS (4), privileged mode (5), DS (6), CS (7), T (8), overflow (10), and current space ID (11 through 15).

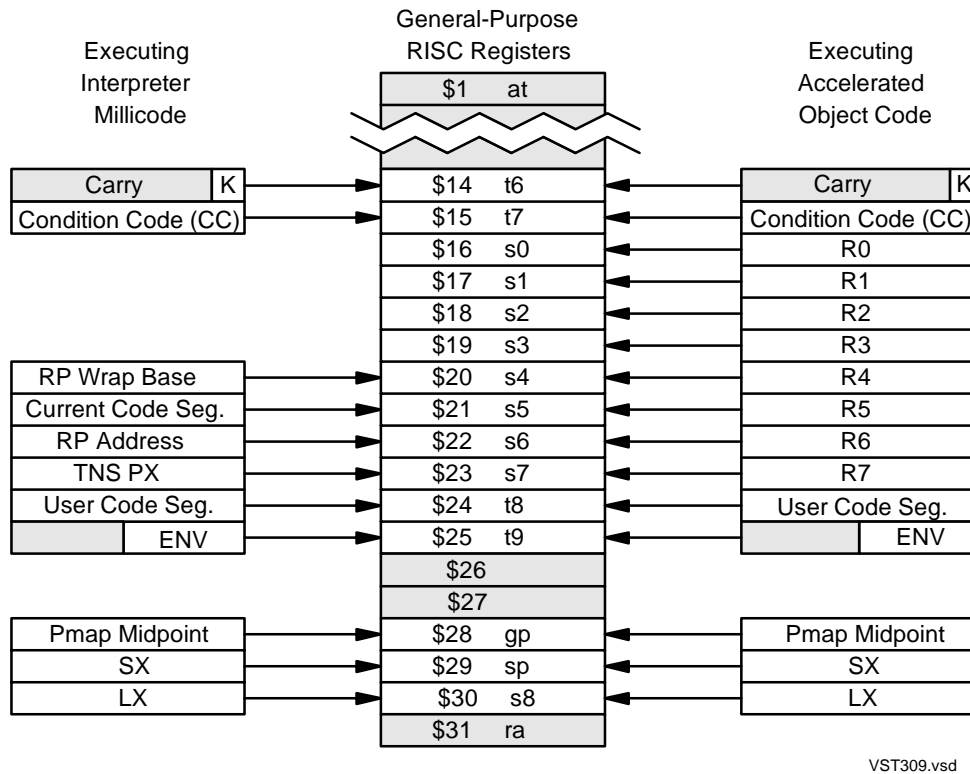
In TNS mode only, \$20 through \$24 contain the following values (all are 32-bit byte addresses). “RP wrap base” is the address of the RP wrap page. “Current code segment” is the address of the base of the current TNS code segment (relative segment 2). “RP address” is the address of the current top-of-stack register; this address is offset a multiple of 512 bytes from the RP wrap base. PX is the current TNS program counter value, the address of the next TNS instruction to be interpreted; because this address is a full 32-bit virtual address, decoding is needed to break it down into a 16-bit word address, a segment number, and a region number (identifying UC, UL, SC, or SL).

In accelerated mode, \$16 through \$23 contain relevant register stack values for R0 through R7. The RISC processor does not use these registers the same way as TNS processors use the register stack. While they nominally represent TNS registers R0 through R7, they contain right-justified values consistent with TNS values only at certain times.

In both modes, \$28 contains the Pmap midpoint address for the current TNS code segment. (The register contains 0 if the current segment is not accelerated.) As described in the previous Pmap discussion, this address provides the basis for converting between TNS P and RISC P values. \$29 and \$30 contain extended (32-bit

byte address) forms of the S and L pointers to the TNS data stack. Because the TNS stack is in segment 0 of region 0, these addresses can be converted to 16-bit word addresses with a one-bit shift. The registers are designated SX and LX.

**Figure 6-44. Most TNS State Values Are Kept in RISC General-Purpose Registers**



# Invoking Privilege for CALLABLE Procedures

Unlike TNS processors (CISC-based), which invoke privilege in the microcode of the calling instruction, RISC-based processors such as NonStop S-series processors, always go through a sequence of events involving an exception to set the privileged state. [Figure 6-45](#) shows the sequences for the three execution modes; the first two are discussed in this topic and the third in [Section 7, Native Execution Mode](#). In all three cases, a program shown on the left side of the illustration is calling the CALLABLE procedure shown at the right side of the figure.

The mechanism is different for CALLABLE transitions in each operating mode and for the special case of a transition to native mode, but each path has some form of the same requirements:

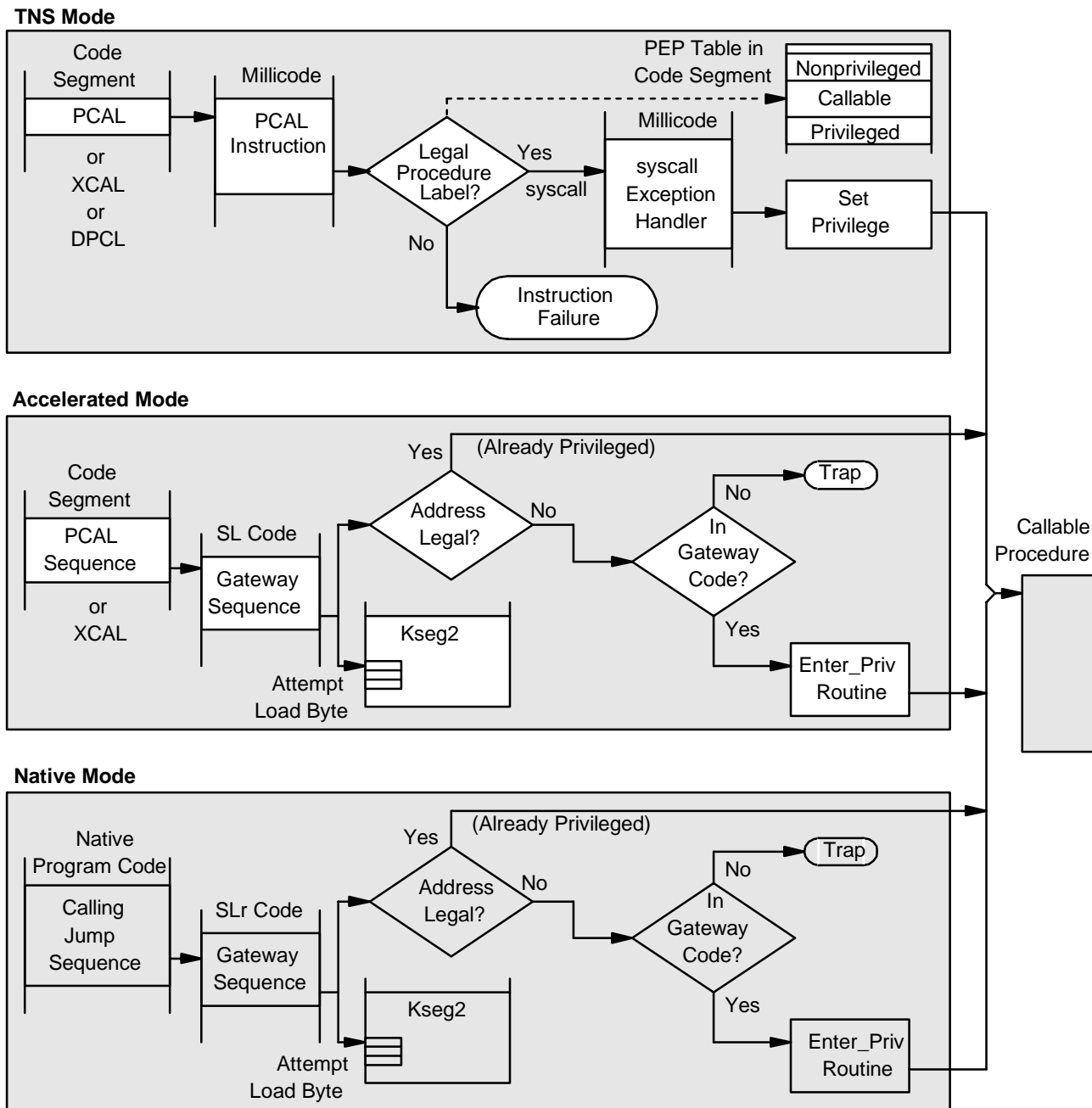
- Verify that this exception arises from an attempt to call a legitimate CALLABLE procedure.
- Ensure that the process state is proper for entering privileged state in the current execution mode. For example, force the stack pointer into a valid range.
- Make any appropriate state changes corresponding to the privilege transition.
- Allow the process to proceed in privileged state.

In TNS mode, when the millicode encounters a procedure call instruction such as PCAL, it first verifies a legal call by comparing the address given as the target procedure with the range of CALLABLE procedure addresses given in the procedure entry point (PEP) table. If the address is legal, the millicode then issues a syscall exception, and the syscall exception handler revalidates that the address is within the legal range, grants the necessary privilege, and transfers to the procedure.

In accelerated mode, the PCAL (or other calling instruction) has been translated to a sequence of RISC instructions. The calling sequence resulting from the translated PCAL invokes a gateway sequence that checks whether the caller is already privileged. To do so, the gateway attempts to load a specific byte in Kseg2 (in the SPAD page). If the gateway can legally load the byte (which is discarded), that means the caller is already privileged, can bypass the succeeding operations, and can transfer to the target CALLABLE procedure.

If, however, the load fails (because the Kseg2 address requires privilege), that means that the caller is unprivileged. This event causes an address error exception, invoking the address error exception handler. This exception handler recognizes that the address being used indicated a request to access a CALLABLE accelerated procedure and verifies that the exception occurred at an address within the designated *gateway* table for an accelerated code region (SL, UL, or UC).



**Figure 6-45. Invoking Privilege Always Requires Taking an Exception**

VST310.vsd



# **7** Native Execution Mode

The topics in this section describe the basic conventions for native-mode execution. The following topics are described.

[Native Mode Uses RISC Register Conventions](#)

[RISC Stack Frames](#)

[Procedure Name Spaces for the System Library](#)

[Example of TNS Call to a Native Library Procedure](#)

[Invoking Privilege Requires Taking an Exception](#)

[Stack Switching for Native Privilege Transition](#)

[Example of Enter\\_Priv Transition](#)

[Far Jumps and Far Gateways Are Needed for SCr](#)

# Native Mode Uses RISC Register Conventions

The NonStop S-series architecture uses the RISC registers in the way that they are defined for the RISC internal architecture. All 32 general-purpose registers, designated GPR[0..31] or \$0..\$31, are used as assigned in the RISC documentation. The following table summarizes these usages.

Names	Registers	Usage	Preservation
	\$0	Constant zero	
at	\$1	Assembler temporary	Temporary
v0, v1	\$2, \$3	Function values	Temporary
a0 through a3	\$4 through \$7	Parameters passed	Temporary
t0 through t9	\$8 through \$15, \$24, \$25	Temporary registers	Temporary
s0 through s8	\$16 through \$23, \$30	Saved registers	Preserved by called procedure
k0, k1	\$26, \$27	Reserved for kernel	
gp	\$28	Global pointer	Preserved by called procedure
sp	\$29	Stack pointer	Preserved by called procedure
ra	\$31	Return address	Preserved by called procedure

The native RISC interprocedure register convention is that the called procedure preserves values in the registers shown in the “Preservation” column. Ordinarily, the s0 through s8 registers and ra are saved (if used by the procedure) and restored, sp is symmetrically decremented and incremented, and gp is unchanged. At a procedure call or return, there is no live information in the registers marked temporary, except for parameters in and values out. (Some millicode interfaces use different conventions.) The temporary registers k0 and k1 are reserved. An interrupt preserves all the registers except k0 and k1.

Two of the registers (sp and ra) are dedicated to the procedure call mechanism. [Figure 7-1](#) shows where these registers (and the global pointer) point within the user space. The sp register designates the tip of the main stack, which is also the low-order byte of the current procedure’s activation record. (See next topic for additional details.) Call-out parameters are stored relative to sp, and call-in parameters and locals are referenced relative to a frame pointer, which is usually sp. The ra register holds the return address for returning to the calling procedure or subprocedure.

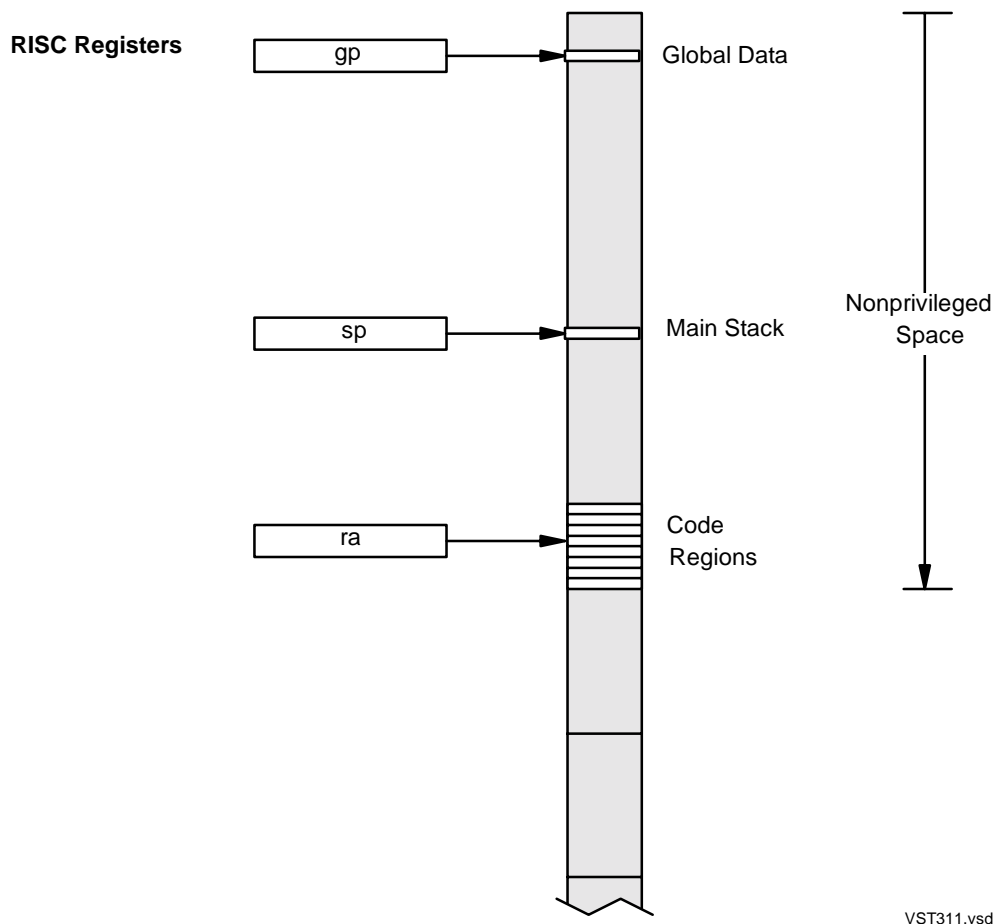
The load and store instructions apply a signed 16-bit offset to the base address in a register, allowing access within one instruction of up to 64 kilobytes of data. Native process global data starts at %h 08000000, so native programs typically set gp to

%h 08007FF0. (Global data is not limited to 64 KB; with two instructions, any address in the 4-gigabyte address space can be accessed.)

Up to four words or parameters can be passed to a procedure in a0 through a3, and results are returned in registers v0 and v1. Procedures save and restore registers s0 through s8 (if they are used) and return sp to its original value. Registers t0 through t9 are used for procedure temporary usage.

Registers k0 and k1 are reserved for kernel exception handlers, and the assembler temporary register (at) is reserved for short-term temporary storage.

**Figure 7-1. RISC Registers Point to Global Data, Main Stack Tip, and Return Address**



# RISC Stack Frames

The primary stacks used by a native process (that is, the main stack and the privileged stack) use RISC stack frame conventions as illustrated in [Figure 7-2](#). These stacks grow downward, toward smaller addresses. They both originate at the high-address end of their containing segments. (In diagrams throughout this manual, addresses are shown as increasing downward on the page.)

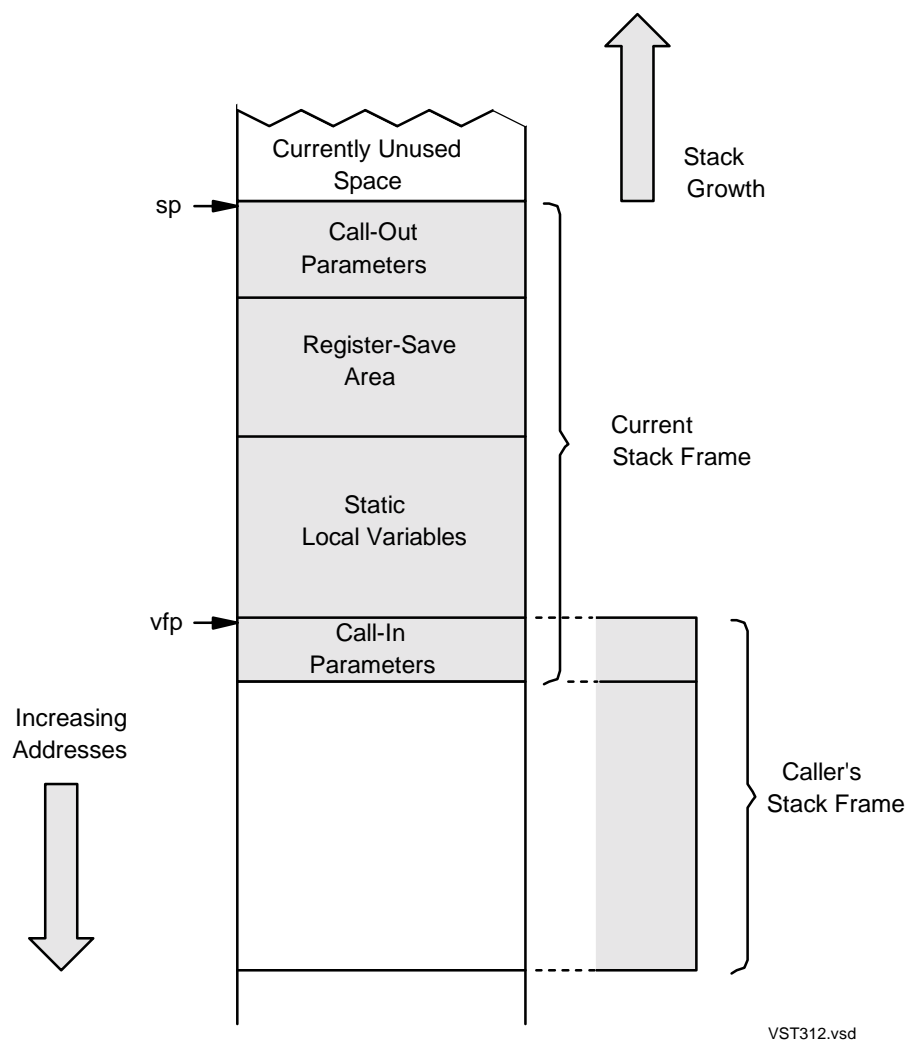
The **sp (stack pointer) register** points at the **stack tip**, which is the lowest-addressed byte in the current stack frame. Typically, sp is static for the frame and, in most cases, also serves as the **frame pointer** for the current frame. (If a procedure has dynamically allocated variables, a separate frame pointer register is used as the static reference point for the frame and sp is adjusted to the new tip.) The frame pointer is used as a base reference to address local variables, saved registers, and formal parameters.

The **virtual frame pointer (vfp)** defines the **frame origin**. The origin is the trailing edge of the stack frame. It coincides with the tip of the frame of the caller. Thus, sp for the caller becomes vfp for the called procedure. No register actually contains the vfp address.

As shown in [Figure 7-2](#), the formal parameters of a procedure call can be considered part of either or both frames—the caller's and the called. The caller's procedure frame is allocated with enough call-out space to hold the parameters for any procedure that it might invoke. Actual parameters are stored into this space by offset references to sp (not pushed, as in TNS frames). The called procedure addresses the formal parameters by offsets from its own frame pointer.

Native stack frames can be increased only in multiples of eight bytes, and sp is rounded up as necessary to accommodate this convention.

Subprocedures use exactly the same stack format as procedures, and sp is adjusted each time any nested subprocedure is called. To accommodate the need for subprocedures to access the outer procedure's local variables, the outer procedure's vfp address is passed as a hidden parameter in the v0 register.

**Figure 7-2. Stack Frames Overlap and Grow to Lower Addresses**

# Procedure Name Spaces for the System Library

The system library exists in two separate **name spaces**, SL and SLr. That is, a name can appear once in each space. The SL name space contains all the TNS library procedures, and the SLr name space contains all the native RISC library procedures.

Most system library procedures are executed as native library procedures in the SLr name space. Native processes can call these procedures by name and invoke them directly. TNS processes, however, cannot invoke these procedures directly, because a transition to native mode is necessary.

So that TNS processes are able to call a target library procedure transparently, using the same name that it would if it were a native process, these two name spaces are bridged by a combination of a **shell map** and special procedures called **to-RISC shells**. Refer to [Figure 7-3](#).

When a TNS or accelerated procedure needs to make a call to a RISC library procedure, it does so by its native name. If an accelerated TNS version of the library procedure exists in the SL name space, the address refers the call to that procedure. This happens in cases where semantics, formal-procedure parameters, or code brevity make it mandatory or more efficient not to switch to the SLr name space and native mode, or when a native version of the target procedure does not exist. For example, there is no native version of the LASTADDRX procedure.

In most cases, however, the required system library procedure is TNS/R native code, which must execute in native mode. Each native procedure that can be called from TNS or accelerated code has an associated to-RISC shell, which is entered as though it were accelerated code but effects the transition to native mode and back.

For calls from TNS code to system library, the interpreter millicode utilizes the shell map to locate the RISC code for the target procedure. This mapping is possible because all the code in SC and SL is accelerated, and all the to-RISC shells start execution in accelerated mode. The shell map is a table located in user space at address 7FFC0000; it is indexed by the low-order 15 bits of any XEP entry whose high-order bit (CS) is 1 (see next topic). The designated map entry holds the starting address of the target procedure; if the procedure is CALLABLE that address designates its gateway in the 7E region.

In an accelerated procedure, the RISC translation of the XCAL instruction does not use the shell map (although DPCL does). The accelerated code jumps directly to the target procedure (or its gateway); the target may be either accelerated code or a to-RISC shell.

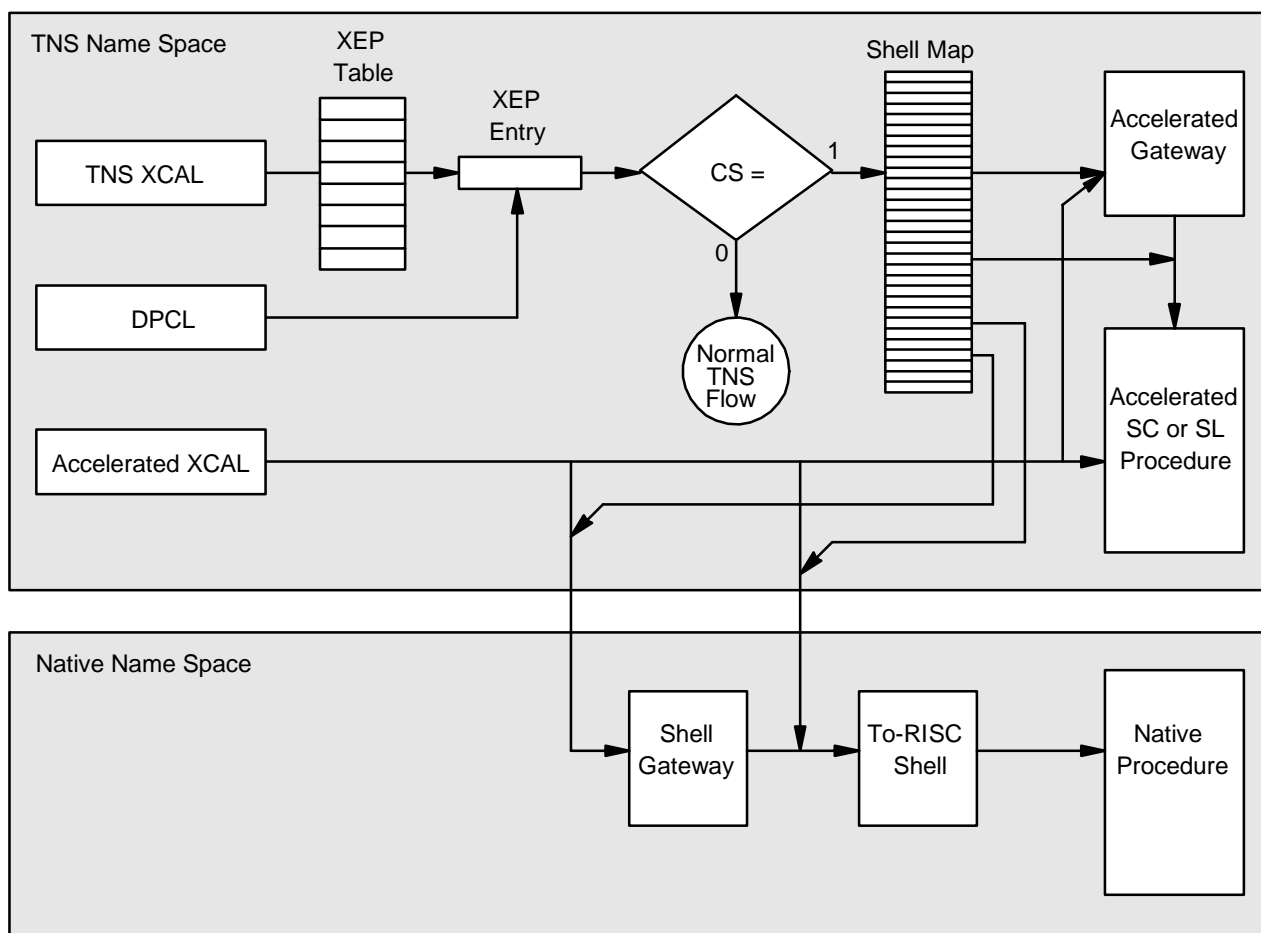
The to-RISC shell invokes millicode to accomplish the transition to native mode, including switching from the TNS stack to the main (RISC) stack. The shell also translates parameters as needed for the particular calling sequence. It then transfers control to the target native library procedure. Upon return, the RISC shell translates



results if needed, reverts to accelerated mode and the TNS stack, and returns control to the TNS or accelerated code that called it.

In summary, referring to [Figure 7-3](#): an XCAL instruction in TNS mode fetches the XEP table entry for the target procedure. A DPCL instruction is presented with the XEP entry as an argument. If CS = 1 in that entry, the shell map is consulted for the starting address of the target RISC code, which may be an accelerated procedure in SC or SL, or may be a to-RISC shell. An accelerated XCAL goes directly to the accelerated code or shell. If the target procedure is CALLABLE, entry is through its gateway.

**Figure 7-3. Gateways and To-RISC Shells Enable TNS Access to System Library**



VST313.vsd

## Example of TNS Call to a Native Library Procedure

The example illustrated in [Figure 7-4](#) assumes that some user code (UC) process is executing in either TNS mode or accelerated mode, using variables on the TNS stack (1). At some point in its execution the process executes an XCAL to a system library procedure (2), such as DNUMOUT. Because DNUMOUT exists as a native mode procedure, the processor must be put into native mode.

The interpreter performs the first part of the XCAL normally, using the XCAL entry number to fetch the XEP table entry. Because CS = 1 in that entry, the interpreter uses the other 15 bits of the entry as an index into the shell map, and fetches the designated address. If that address is zero, the XEP entry was invalid; otherwise it designates either a to-RISC shell or an accelerated procedure.

The low-order bit of the shell map entry distinguishes the two valid cases: if it is odd, this address -1 designates an accelerated procedure in the SC or SL part of the system library; the interpreter switches to accelerated mode and jumps to that code. For accelerated procedures, such as LASTADDRX, the address is in the shell map position indicated by bits <1:15> of the normal XEP entry for the TNS code of that procedure.

An even address designates a to-RISC shell, so the interpreter puts the process into accelerated mode and calls the shell. Shells are assigned arbitrary XEP entry values corresponding to nonexistent TNS code segments, such as SC.00 through SC.04.

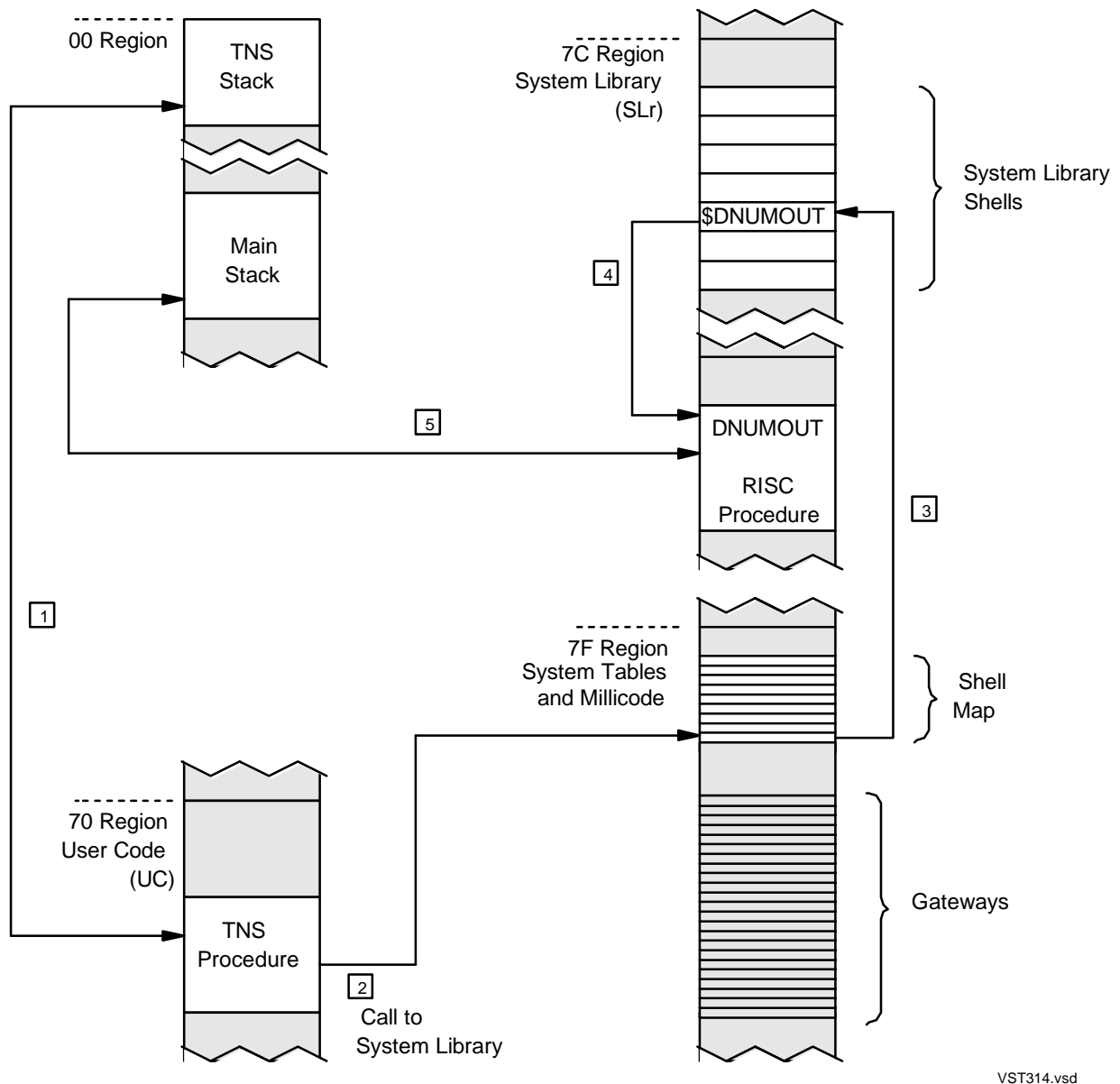
In this case, the address in the shell map transfers control (3) to the to-RISC shell called \$DNUMOUT in the RISC portion of the system library (SLr in the 7C region).

The to-RISC shell performs several functions. It switches the processor mode to native mode. It saves the TNS return information in a special stack frame created on the appropriate RISC stack (either the main stack or the privileged stack) and switches the stack pointer to that frame. Then the RISC shell calls the entry point of the called RISC procedure (6). (The procedure generally has the same name that the TNS caller used to invoke it.) The RISC procedure now executes in native mode, using the main stack or the privileged stack (7).

The shell chooses the privileged stack if the called procedure runs privileged, because either the shell is callable or the calling TNS procedure is already privileged.

Upon completion of the called procedure, the procedure exits normally to the procedure that called it, the to-RISC shell. The shell restores TNS values to appropriate registers, sets the process mode to accelerated or TNS, and returns to the calling procedure through the saved return information.

When the original call was TNS code, as in our example, the shell returns into the code interpreter in millicode, which resumes execution of the original code stream following the XCAL.

**Figure 7-4. A TNS Call to the System Library Is Directed Through the Shell Map**

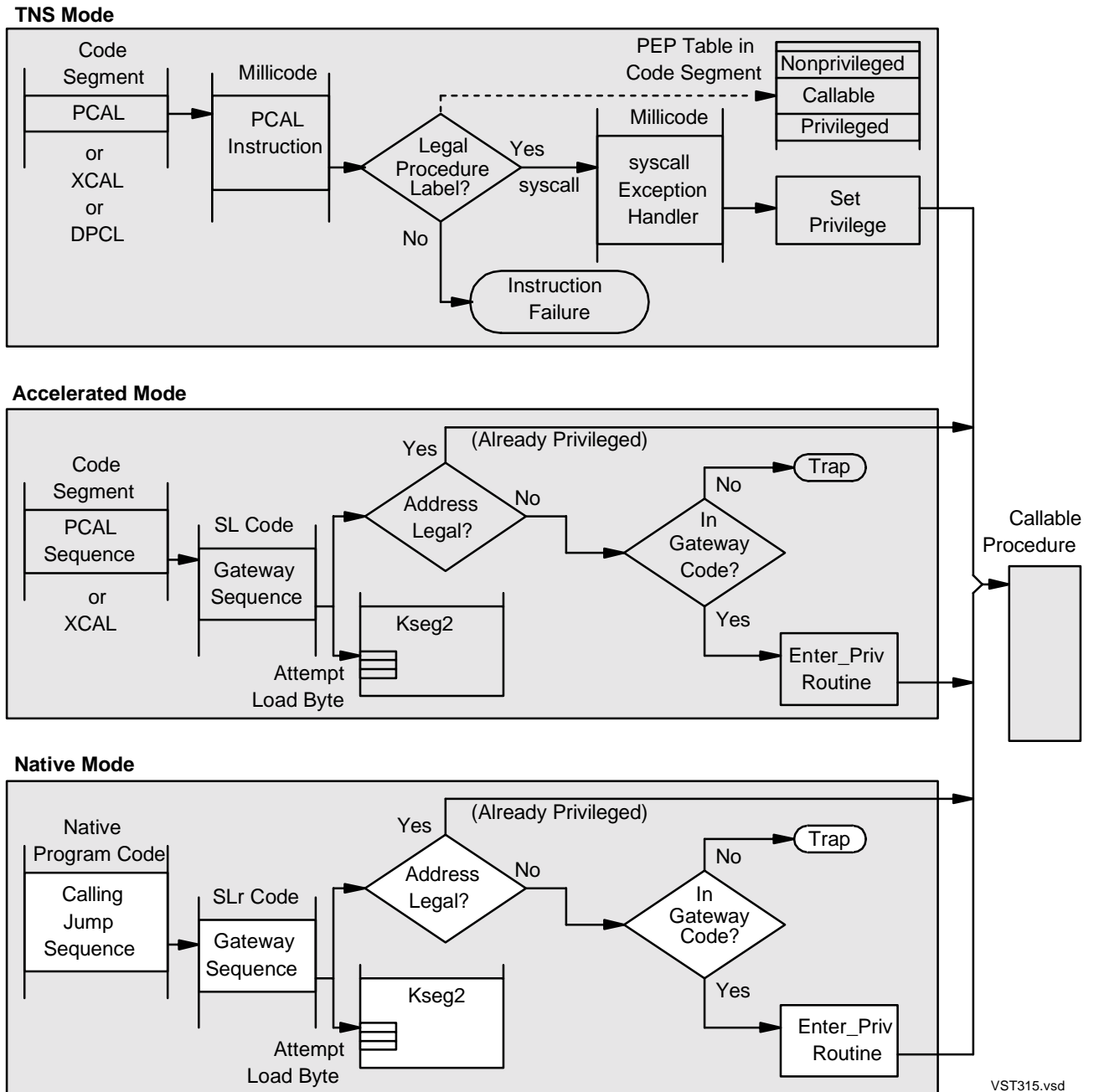
# Invoking Privilege Requires Taking an Exception

Unlike TNS processors (CISC-based), which invoke privilege in the microcode of the calling instruction, RISC-based processors, such as the NonStop S-series processors, always go through a sequence of events involving an exception to set the privileged state. [Figure 7-5](#) illustrates the sequences for the three execution modes; the first two are discussed in [Section 6, TNS Execution Modes](#), and the third in this topic. In all three cases, a program shown on the left side of the figure is calling the CALLABLE procedure shown at the right side of the figure.

The native mode case is similar to the accelerated mode case. That is, the calling code first invokes a gateway that loads a byte from Kseg2. However, the address of the target byte is different, indicating a native Enter\_Priv transition. The byte is loaded to register v0, not discarded (its negative value indicates to the called procedure that it was called from a privileged procedure). If the test is successful, the exception handler moves the stack pointer to a fixed address on the privileged stack; it puts the original stack pointer into v0 so that the called procedure can find its original parameters. Then it allows the process to proceed in privileged state.

To-RISC shells for CALLABLE native procedures are themselves CALLABLE and use the same gateways that native procedures use. Yet another Kseg2 address identifies these gateways, which are entered in accelerated mode and do not switch stacks (the forthcoming Enter\_RISC mode transition does that).

(Gateway tables are like CALLABLE PEP entries: the operating system will not run a program containing one unless the program is licensed or initiated by the super ID, 255,255.)

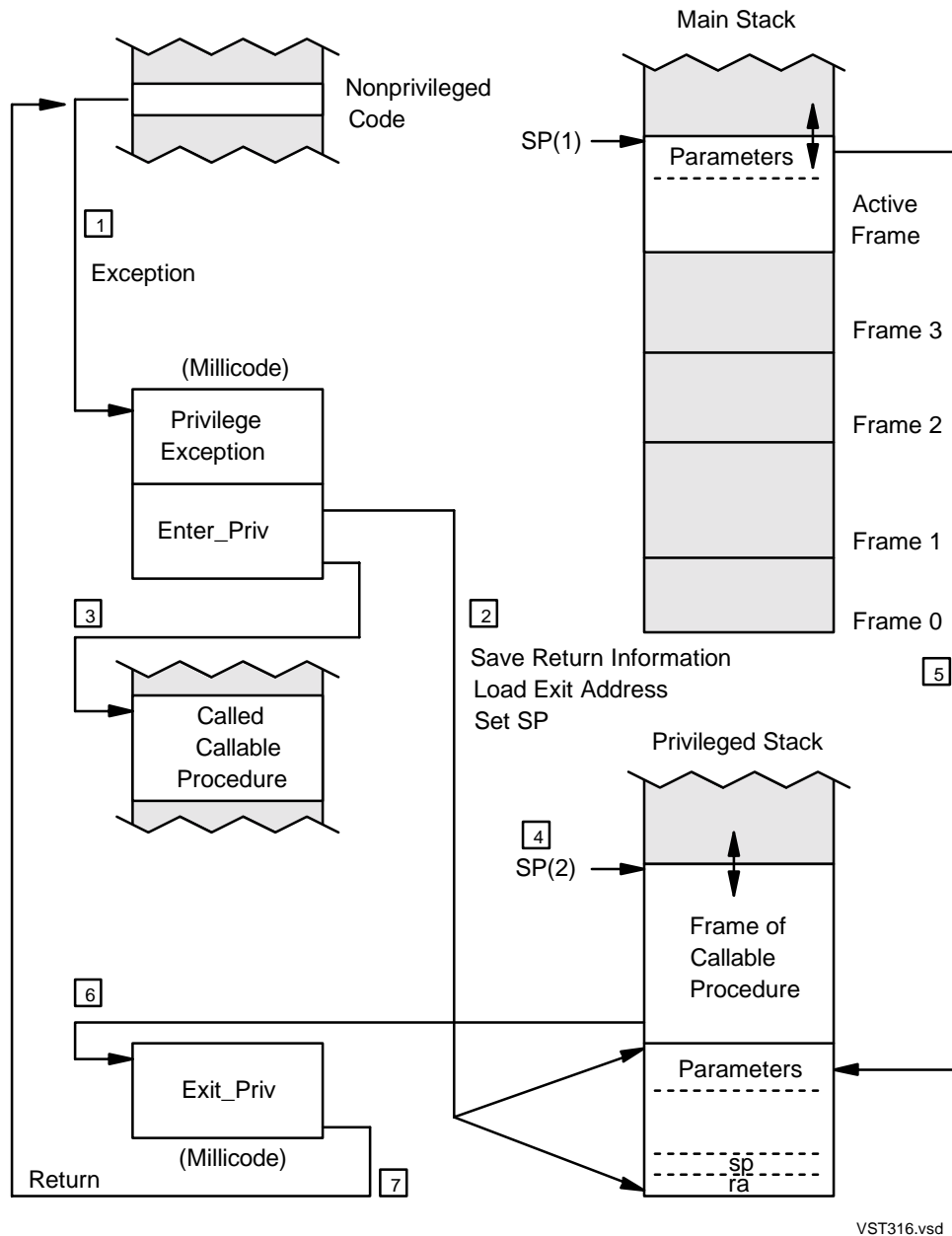
**Figure 7-5. syscall Exception or Address Error Exception Sets Privileged State**

# Stack Switching for Native Privilege Transition

If a nonprivileged process, operating in nonprivileged native mode, calls a callable procedure, the process must switch its operation from the main stack to the privileged stack. This switch is performed as part of the transition to privileged mode (discussed in [Invoking Privilege Requires Taking an Exception](#) on page 7-10).

The stack switch is accomplished by these steps. The step numbers correspond to the numbered callouts in [Figure 7-6](#).

1. The call in nonprivileged code to a callable procedure causes a privilege exception, invoking the exception handler in millicode. (See [Invoking Privilege Requires Taking an Exception](#) on page 7-10.)
2. Upon verifying this exception as an invocation of a legitimate callable procedure, the exception handler passes control to the Enter\_Priv routine. This routine builds a special stack frame on the privileged stack, beginning with the caller's return address (ra) and existing stack pointer address (sp). These items are followed with space for the maximum number of arguments.
3. The Enter\_Priv routine now moves sp to the privileged stack (above the parameter area) and invokes the called callable procedure. The return address in register ra now designates the Exit\_Priv millicode for use in Step 6.
4. The called procedure allocates its frame on the privileged stack.
5. The called procedure transfers parameters from the caller's frame on the main stack to the new frame on the privileged stack. (Parameters passed in registers a0 through a3 are undisturbed by Enter\_Priv and so need no copying.) The procedure executes on the privileged stack until ready to return, placing any return values in processor registers.
6. The called procedure exits normally to the address it finds in the ra register. Because this register was set to point at the Exit\_Priv routine (Step 2), control passes to that routine.
7. The Exit\_Priv routine switches the mode back to nonprivileged and retrieves the ra and sp values that were stored at the beginning of the privileged stack. These are the caller's values. Loading these values into the ra and sp registers, the millicode returns control to the caller. Any values returned in v0 and v1 by the callable procedure are undisturbed by the Exit\_Priv operation.

**Figure 7-6. Millicode Routines Switch Between Main and Privileged Stacks**

## Example of Enter\_Priv Transition

Because privileged mode operation provides access to operating system elements, transitions to privileged mode by nonprivileged procedures must be restricted to callable procedures only. Transitions in the reverse direction, from privileged to nonprivileged, occur while the code is running privileged (and therefore trusted), and so this transition direction is accomplished simply by the Exit\_Priv routine.

The transition to privileged mode, for a legitimate call to a callable procedure, is accomplished by a set of short code sequences called **gateways**, which are located in specific memory areas called **gateway tables**. Gateway tables are located all together in region 7E. User code (UC, UCr) and user library (UL) seldom have such callable procedures and so rarely have gateway tables. In fact, most callable procedures are in the system library area. The example shown in [Figure 7-7](#) illustrates the gateway table for the RISC system library, which resides in user space.

Note the major elements of the example shown. These are the calling instructions in the user code (on the left), the called procedure code in the native code part of the system library (SLr in the 7C region), the gateway table, and a scratchpad page (SPAD) in Kseg2. The example assumes the procedure being called is the READX procedure.

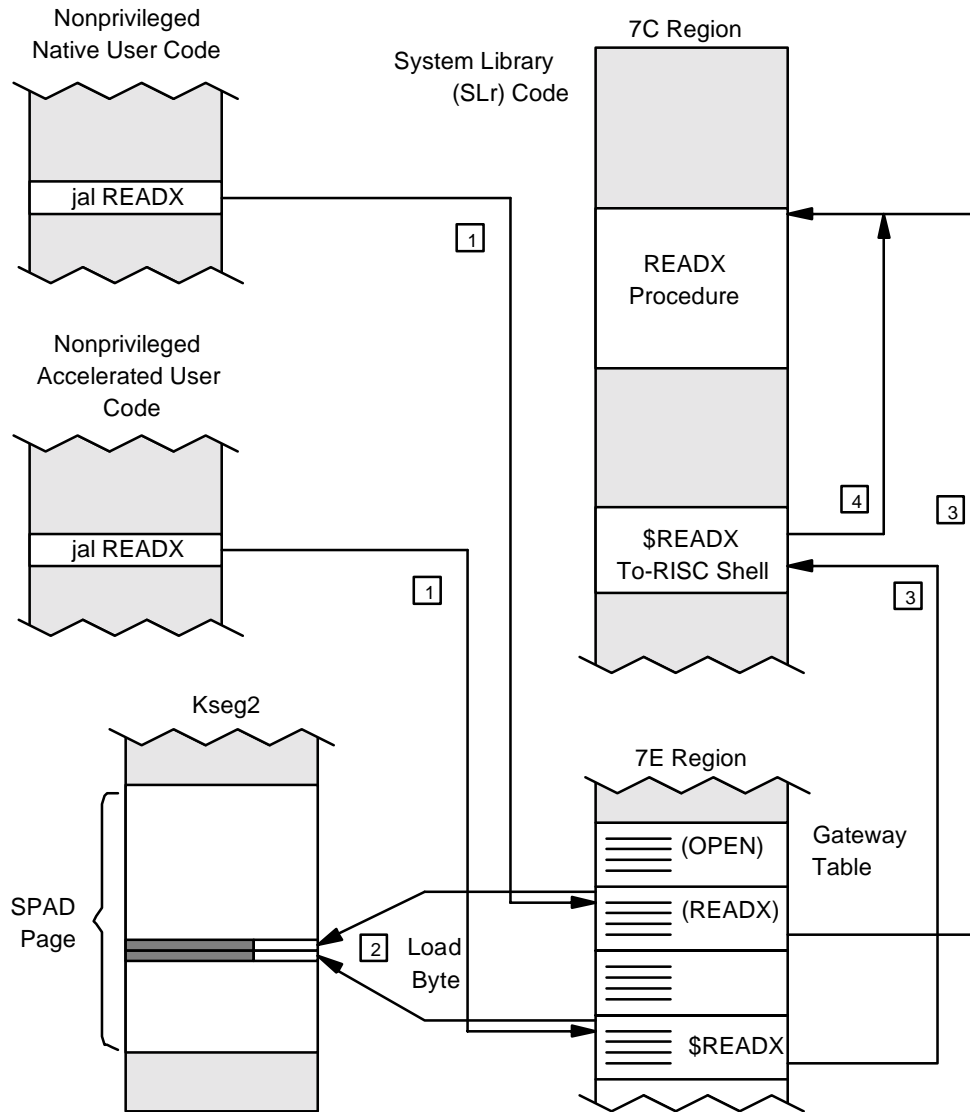
Instead of jumping directly to the entry point of the called procedure, the caller goes to the gateway for the called procedure (1). The gateway consists of a sequence of RISC instructions, one of which is a load-byte instruction (2) that attempts to load a certain byte of the scratchpad page (SPAD), which is located in privileged virtual memory (Kseg2). Such a reference requires privileged mode. For the privileged caller, the load occurs and the jump to the procedure entry point immediately follows. But for the nonprivileged caller, an address error exception occurs. This invokes the exception handler, which (recognizing that the address is special) checks the call for validity and allows the process to proceed in privileged state.

The final instruction in the gateway is the jump to the entry point of the called procedure (3). The called procedure now executes in privileged mode and performs the requested operation.

An accelerated caller invokes the RISC part of a to-RISC shell by way of a very similar gateway mechanism. The address of the load-byte instruction is different, to indicate that the Enter\_Priv transition is to a shell, so (a) it takes place in accelerated mode, (b) zero instead of **sp** is stored in the special transition frame, and (c) the shell rather than Enter\_Priv moves **sp** to the privileged stack. The same sequence of steps occurs, with the third jumping to the shell, which ultimately (4) calls the target RISC procedure.

Essentially the same three-step sequence occurs when an accelerated TNS procedure calls READX and is diverted to the gateway for \$READX, the shell. (4) \$READX calls the same native READX procedure.



**Figure 7-7. Deliberate Invocation of Error Exception Triggers Privilege Transition**

VST317.vsd

# Far Jumps and Far Gateways Are Needed for SCr

In computing a 32-bit PC address from the 26-bit target field of a RISC jump instruction, the RISC processor takes the four high-order bits from the current program counter value. That fact results in 16 possible **direct jump areas** in the 4-GB virtual address space, each being 256 megabytes. Most process code (UCr and SLr), as well as the interpreter and nonprivileged instruction set millicode, are in the last direct jump area of user space. System code (SCr) and some privileged millicode, however, are in the first direct jump area of kernel space. Jumps between these two areas are accomplished by means of far jumps.

A **far jump** is an entry in a **far jump table** consisting of four or more RISC instructions. These instructions use a full 32-bit target address to enable crossing the boundaries of direct jump areas.

There are only two far jump tables for native code, each combined with gateway tables—one in the 7E region and one at the end of SCr. (SCr has a gateway table, but its gateways are degenerate because the process is already in privileged state when running in SCr.) All calls to SCr, whether they originate in UCr, an SRL, or SLr, must be routed through the far jump table in the gateway area. Any calls in the reverse direction, from SCr, are only to SLr in user space. (Far jumps to millicode are placed with the accelerated far jumps and gateways at the ends of SC and SL as described in Section 6.)

[Figure 7-8](#) illustrates five common cases of calls between the three main kinds of code areas: user code (UCr), SLr, and SCr. Three of the calls originate in user code (calls to A, B, and D), one in SLr (call to C), and one originates in system code (another call to B). The labels FJ, GW, and GW + FJ designate whether the entry is a far jump table entry, a gateway table entry, or a combined entry, respectively.

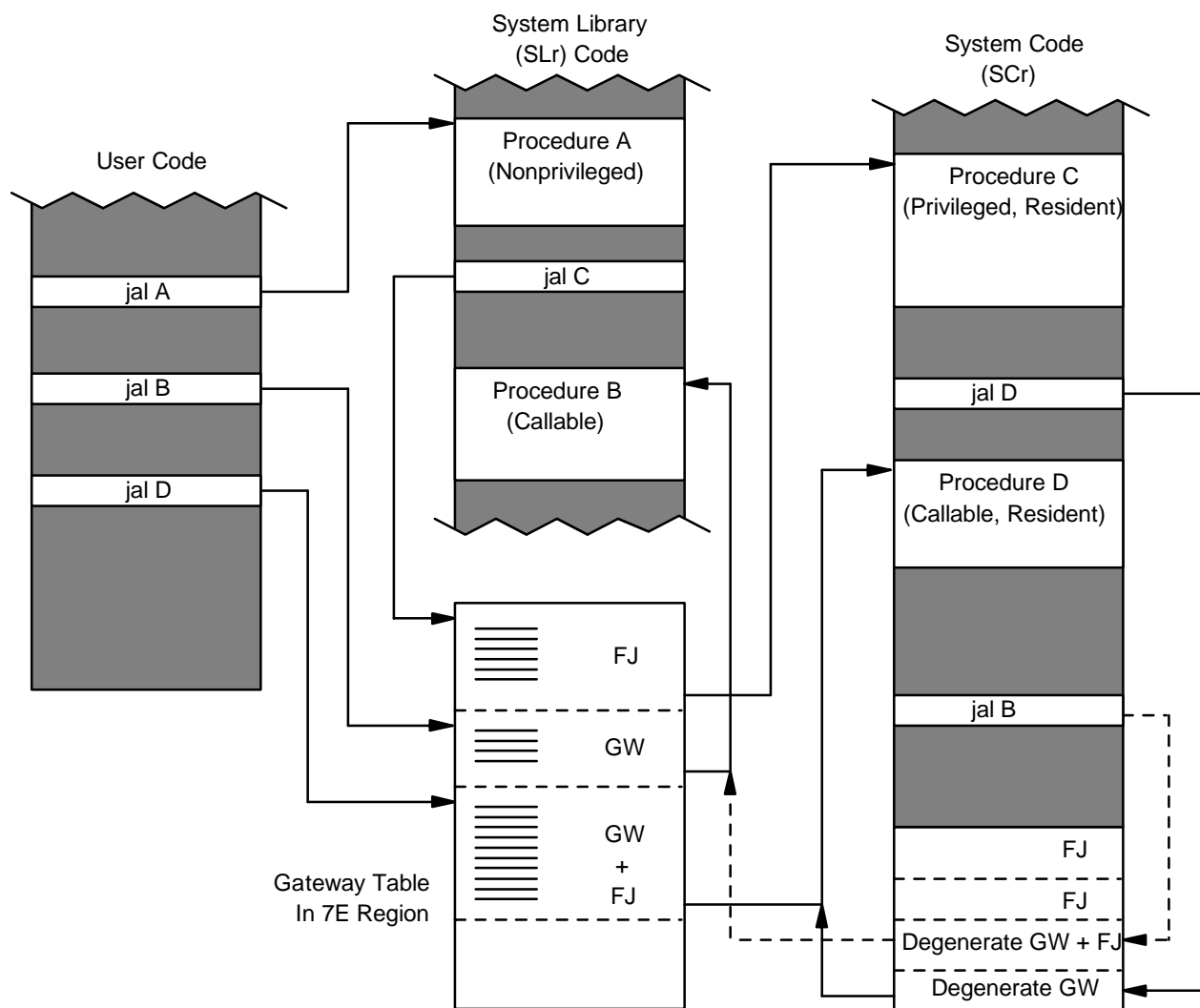
Procedure A is a nonprivileged procedure located in SLr. Because it requires no privileged mode transition, it requires no gateway table entry, and because it is in the same direct jump area as the example call, it requires no far jump table entry.

Procedure B is a callable procedure in SLr, and therefore requires a gateway table entry. When the procedure is called from user code, the call must jump first to the gateway table entry. As described in the preceding topic, this indirect access makes the privileged-mode transition before jumping to the procedure. However, when B is called from SCr (dashed lines), the first jump is to the degenerate far gateway table entry in SCr, which contains a jump-register instruction that jumps directly to B. The degenerate gateway entry involves no privilege transition because the call is made from privileged code, but must set a register to indicate that the caller was privileged and no parameter copying is needed.

Procedure C is a privileged (but not callable) procedure located in SCr. It can be called only by privileged code and therefore requires no gateway table entry. When it is called from SLr, the call jumps first to the far jump table entry for procedure C; the final instruction of this entry completes the jump to procedure C code in SCr.

Procedure D is both callable (requiring a gateway table entry) and located in SCr (requiring a far jump table entry). Both entries are combined as a **far gateway**, concluding with the jump instruction that completes the jump to procedure D. Calls to D from within SCr go through a dummy gateway in SCr.

**Figure 7-8. Far Jump Tables Allow Calls to Cross Direct Jump Area Boundaries**



VST318.vsd



---

---

# 8

---

---

# Interrupt System

Certain defined events can interrupt the processing of a running program. Servicing of these events is accomplished by interrupt handlers, which are not processes. This section describes how control is passed to and from the interrupt handlers. The topics are as follows:

[Interrupt Overview](#)

[Interrupt Sequence](#)

[Interrupt Stack Marker Format](#)

[Transferring Control to an Interrupt Handler](#)

[Interrupt Masking](#)

[TNS Interrupts](#)

# Interrupt Overview

[Figure 8-1](#) is a simplified overview of the sequence by which an **interrupt** condition temporarily stops the execution of the current process (or possibly some other interrupt handler), executes the **interrupt handler** that is appropriate for the particular interrupt condition, and eventually restores control to the interrupted program code.

Step 1 assumes that there is some code currently executing.

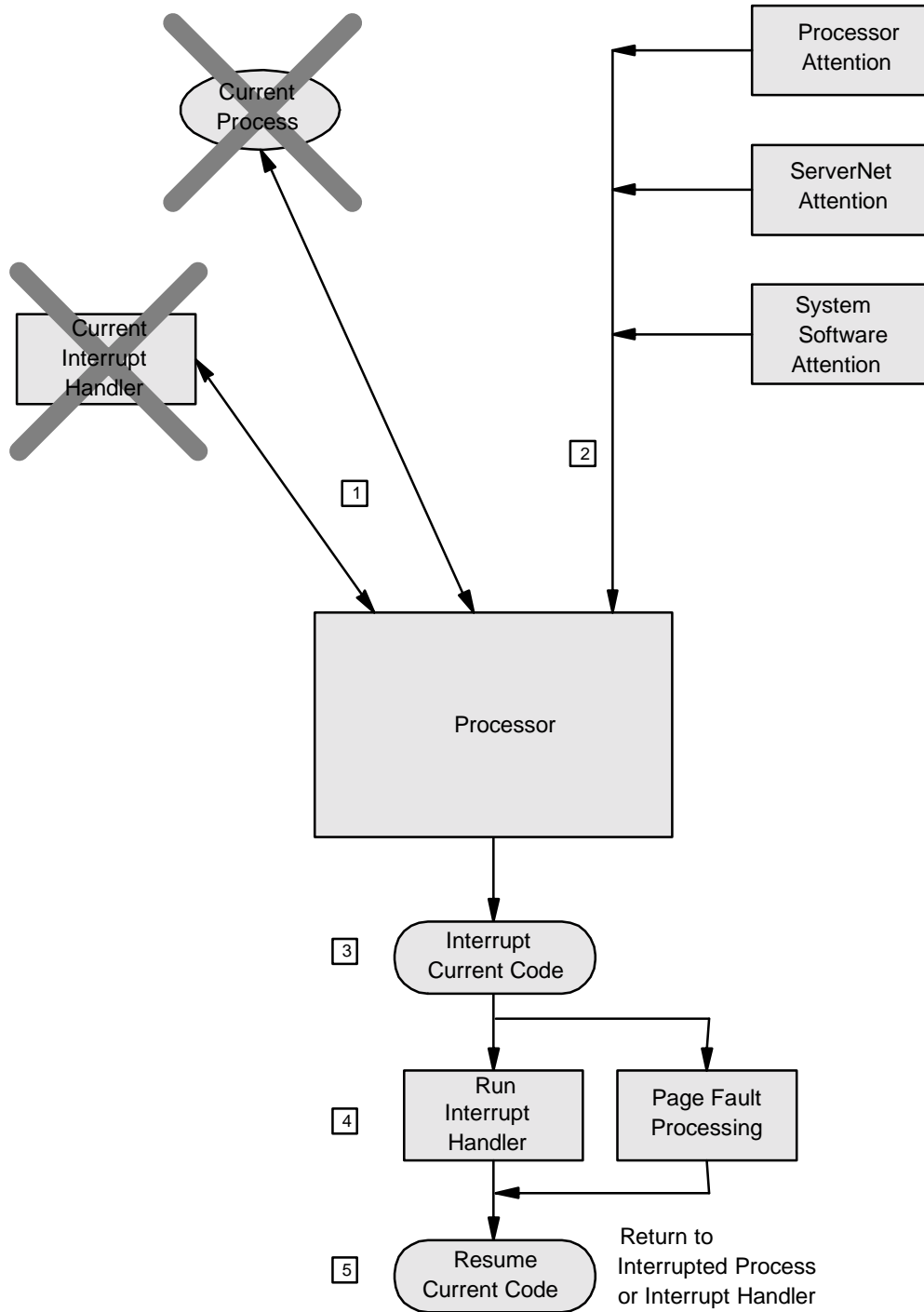
In Step 2, some interrupt condition occurs. As indicated in [Figure 8-1](#), such interrupt conditions are generally grouped into four categories, each processed a little differently. Although there are other forms of interrupts than those identified in the figure, such interrupts are handled within the RISC chip or in the millicode, and do not require processing by an interrupt handler.

One category comprises those conditions for which the processor requires attention, such as arithmetic overflow condition. Another category is for ServerNet attention, such as the arrive of an incoming packet through the ServerNet hardware into an interrupt queue. Still another category is for system software attention, such as to dispatch a process.

Many interrupts do not get processed immediately. For others, the processor evaluates incoming interrupts on the basis of priority and also masking (some interrupts can prohibit processing of certain other interrupts until they themselves have been serviced). Other interrupts of a particular kind are queued (first in, first out) before they even get considered for processing.

Eventually, however, a given interrupt is allowed to interrupt the code of the current program (Step 3), and the processor begins the execution of the appropriate interrupt handler (Step 4). In the case of a page fault, an interrupt does not occur; instead, the page fault is handled by system code in the process environment.

Finally, after the interrupt handler has run to completion, it passes control back to the interrupted code (Step 5)—that is, assuming that the interrupted code still has highest execution priority. Conceivably, some other code may have achieved higher priority during the time that the interrupt handler was running, and the interrupted code must wait its turn.

**Figure 8-1. An Interrupt Can Interrupt a Process or Another Interrupt Handler**

VST360.vsd

# Interrupt Sequence

[Figure 8-2](#) expands on the simplified interrupt sequence shown in the preceding topic. This diagram separates hardware and millicode elements (top part) from the software elements (lower part).

The focal point for the various categories of interrupts is the Cause register in the RISC chip. The shaded part of the Cause register represents “internal interrupts,” and the unshaded part is used for “external interrupts.” The sequence is as follows.

When the processor millicode detects an interrupt condition from system code (1), or if it detects a processing error, it records that information in a simulated interrupt register (INTA) for later access by the interrupt handlers and sets a bit in the RISC Cause register (2), if that bit is not already set.

On the other hand, if the interrupt condition comes from the ServerNet hardware, the access validation and translation (AVT) hardware first puts an **interrupt packet** into one of four dedicated **interrupt queues** (3). The queues are for IPC (interprocessor communications) interrupts, I/O interrupts, error interrupts from either IPC or I/O, and ServerNet coherency interrupts. When such interrupt packets are put into a queue, the AVT sets a bit in an associated Cause register, which in turn sets one of the external interrupt bits in the RISC Cause register (4).

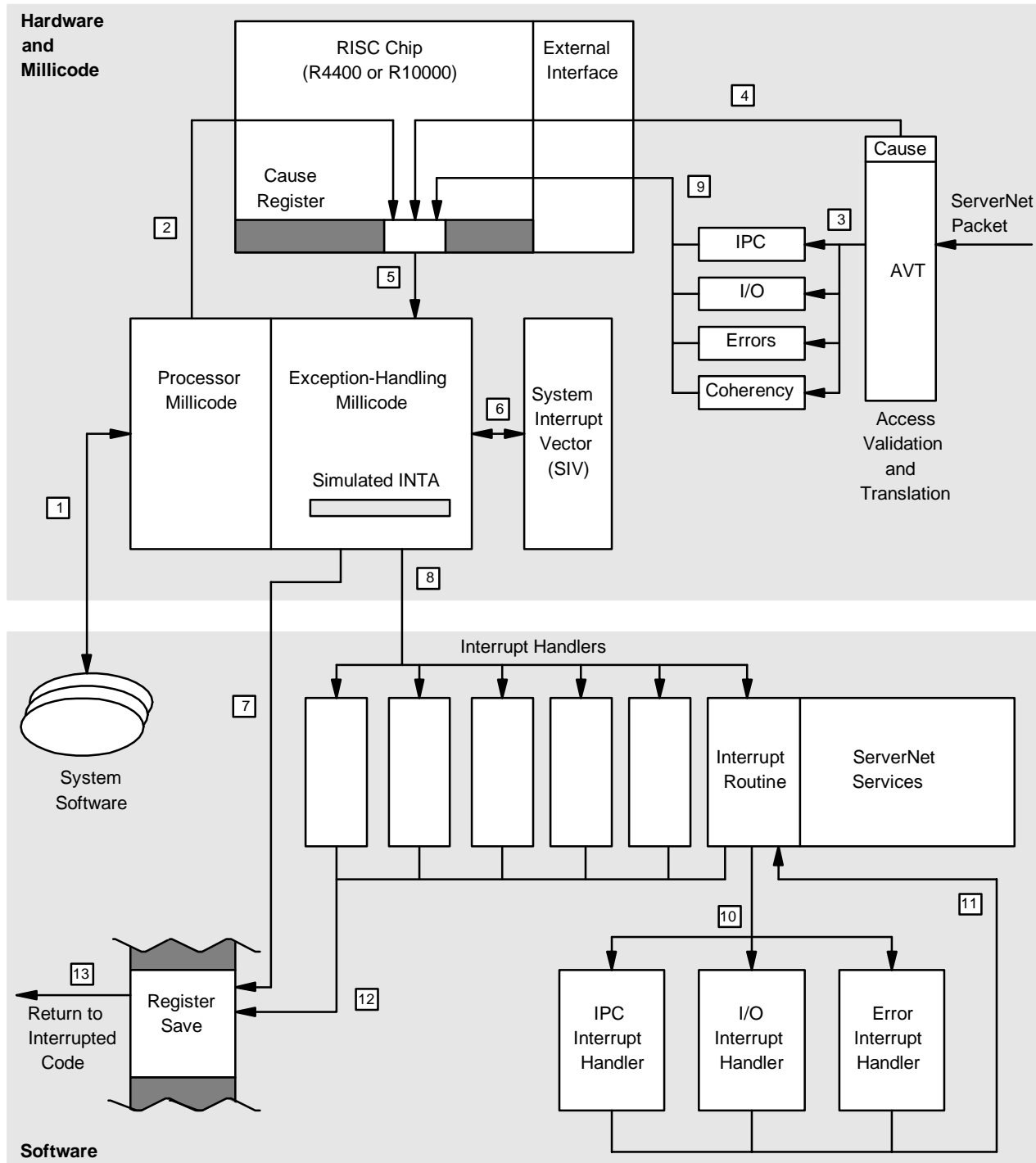
The interrupt bit in the RISC Cause register causes an exception to occur, which causes the processor to transfer control to the exception-handling millicode (5). The exception-handling millicode either fully handles the interrupt itself (such as for coherency) or it passes control to software. For software processing of interrupts, the millicode fetches the appropriate entry from the system interrupt vector (SIV) (6), and saves the environment of the interrupted code in a stack marker in the register-save area (7). Using information from the SIV entry, the millicode transfers control to the appropriate software interrupt handler (8).

In the case of ServerNet interrupts, an additional step is necessary, in which ServerNet services picks out interrupt packets from the interrupt queues (9). An interrupt routine that is part of ServerNet services maintains read pointers into the four queues, which try to catch up to the write pointers maintained by the AVT as it puts more interrupt packets into the queues. That is, the interrupt routine tries to empty the queues by reading and servicing the interrupt packets as fast as the AVT can add more entries. Reading and writing of the queues are done circularly.

The ServerNet services interrupt routine passes the information in each interrupt packet to one of three interrupt handlers (10), as indicated. When the interrupt handler finishes, it returns to ServerNet services (11).

Once the interrupt handler (or ServerNet services interrupt routine) completes execution, it restores the environment of the interrupted code from the register-save area (12) and returns to the interrupted code (13).



**Figure 8-2. Interrupts Invoke Interrupt Handlers, Return to Interrupted Code**

VST361.vsd

# Interrupt Stack Marker Format

Beginning with the G05.00 RVU, all code is native, and therefore the process environment is saved in the register-save area. [Figure 8-3](#) illustrates the arrangement for storing the environment only for the case of TNS code.

The interrupt stack frames are allocated in the system data segment, each one corresponding to one of the interrupt handlers. When an interrupt occurs, the interrupted environment is saved in the stack marker of the appropriate stack frame.

The particular stack frame for a given type of interrupt is addressed by the **LX register** (a RISC general-purpose register that contains a 32-bit byte address). The address resides in the system interrupt vector and is referred to as LX<sub>i</sub>, where “LX” is the byte address of the first L location in the system data segment and “i” is the interrupt number (an index).

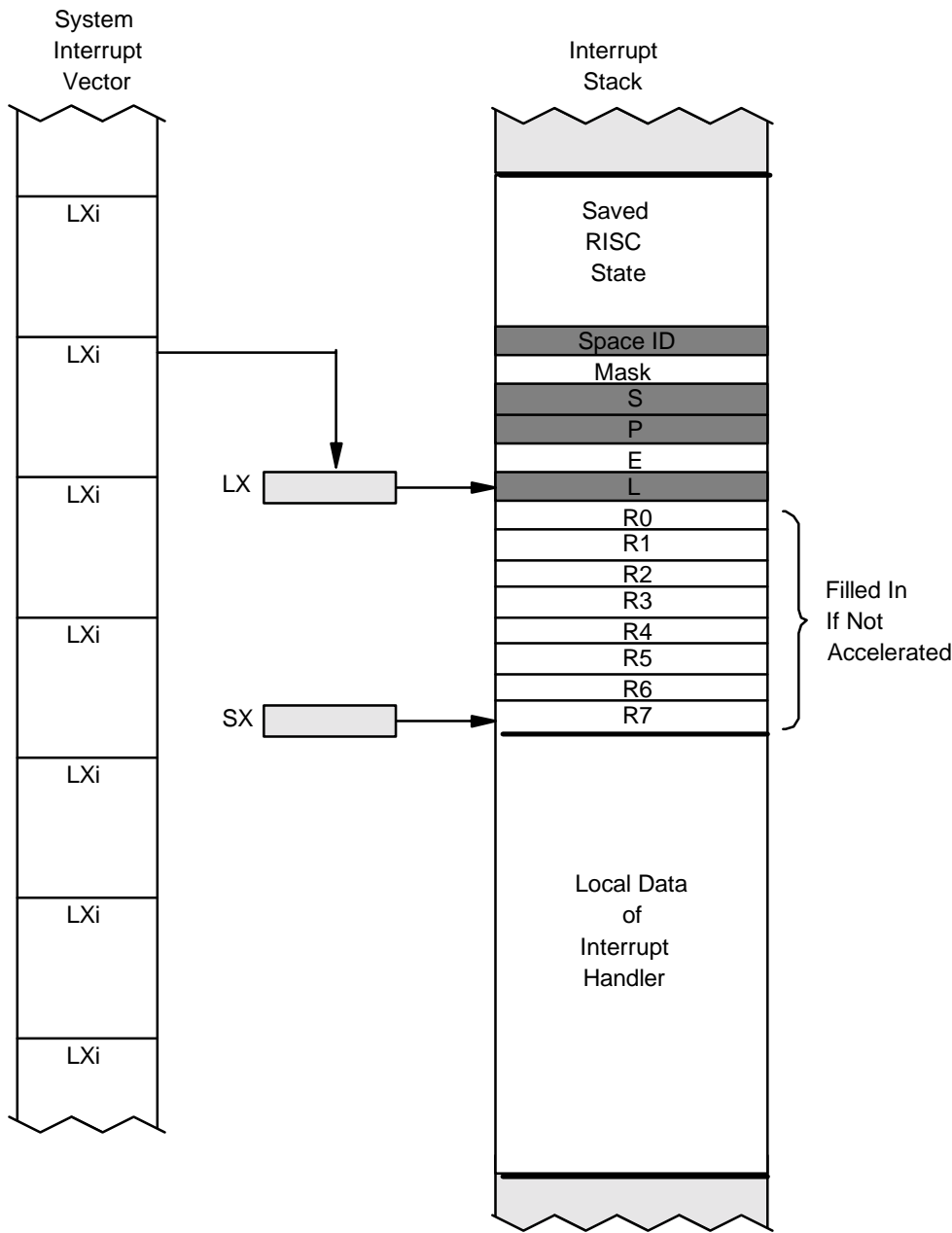
The interrupt stack marker provides sufficient space to store both the RISC environment and the TNS environment of the code that was currently executing. Although the stack marker has a fixed format, individual slots might not be filled in all situations. Only the information actually needed to restore the preinterrupt state is actually stored.

The RISC environment consists of the 31 general-purpose registers (GPR 0 always contains the value 0 and so never needs to be saved), plus the arithmetic high and low registers, two program counter values, the current mode (accelerated or nonaccelerated), and other information. Of the two program counter values, one is the address at which the interrupt occurred (or the address that caused the interrupt); the other is the address where execution of the process (or other interrupt handler) is normally expected to resume. (These values are often identical.)

In addition to storage space for RISC state, the stack marker also provides 14 TNS-word entries for TNS information. These entries are allocated in the same sequence as in CISC TNS processors, but four entries (space ID, S, P, and L) are never used. Only the Mask and Environment register values are saved—and they are always saved whenever the mode is accelerated or TNS. Of the Environment register bits, only PRIV and DS (bits 5 and 6) are ever saved. Conversely, the register stack values (R0 through R7) are saved only in nonaccelerated mode.

The **SX register** is set to contain the 32-bit byte address of the final location of the stack marker, which is the top of the interrupt stack when the interrupt handler begins execution.

**Figure 8-3. The Interrupt Stack Marker Saves RISC and TNS State**



VST362.vsd

# Transferring Control to an Interrupt Handler

After the interrupted environment has been saved, as described in the preceding topic, the interrupt millicode transfers control to the interrupt handler in system software. To do this transfer, the millicode refers to a table of 32-byte items in system data called the system interrupt vector (SIV) and selects the appropriate entry. See [Figure 8-4](#).

To start the interrupt handler, the following steps occur, all involving SIV fields.

1. The E value of the SIV entry is loaded into the Environment register. E defines the interrupt handler's environment. Typically it contains a hexadecimal value of 705 or 706: that is, privileged, system code, system data, and system code segment number 5 or 6.
2. LX<sub>i</sub> is loaded into the RISC general-purpose register that is used as L. LX<sub>i</sub> is configured to contain the byte address of the "base" of the interrupt stack for this particular interrupt. Specifically, it points at the (unused) L value location in the stack marker (see preceding topic).
3. SX is set to LX + 16 bytes (that is, eight TNS words higher than LX).
4. The RISC program counter is set with the address PX<sub>i</sub>. PX<sub>i</sub> is the accelerated code starting address of the interrupt handler.
5. The Mi value in the SIV entry is ANDed with the current Mask A register setting to derive a new setting for the Mask A register. Mi is a mask value for masking off unwanted interrupts while the handler for this particular interrupt executes.

The SIV entry also contains two fields that are needed for special purposes:

- Pmap contains the byte address that points to the midpoint of the Pmap for the TNS code segment containing the interrupt handler. The Pmap information is needed for returning to the correct address whenever the interrupt handler calls procedures or subprocedures.
- The next two words are for passing parameters to the interrupt handler. The low-order 16 bits in each word usually contain the 16-bit parameter that would be passed to the interrupt handler in any TNS processor, and the high-order 16 bits, if used, contain additional special parameter information.

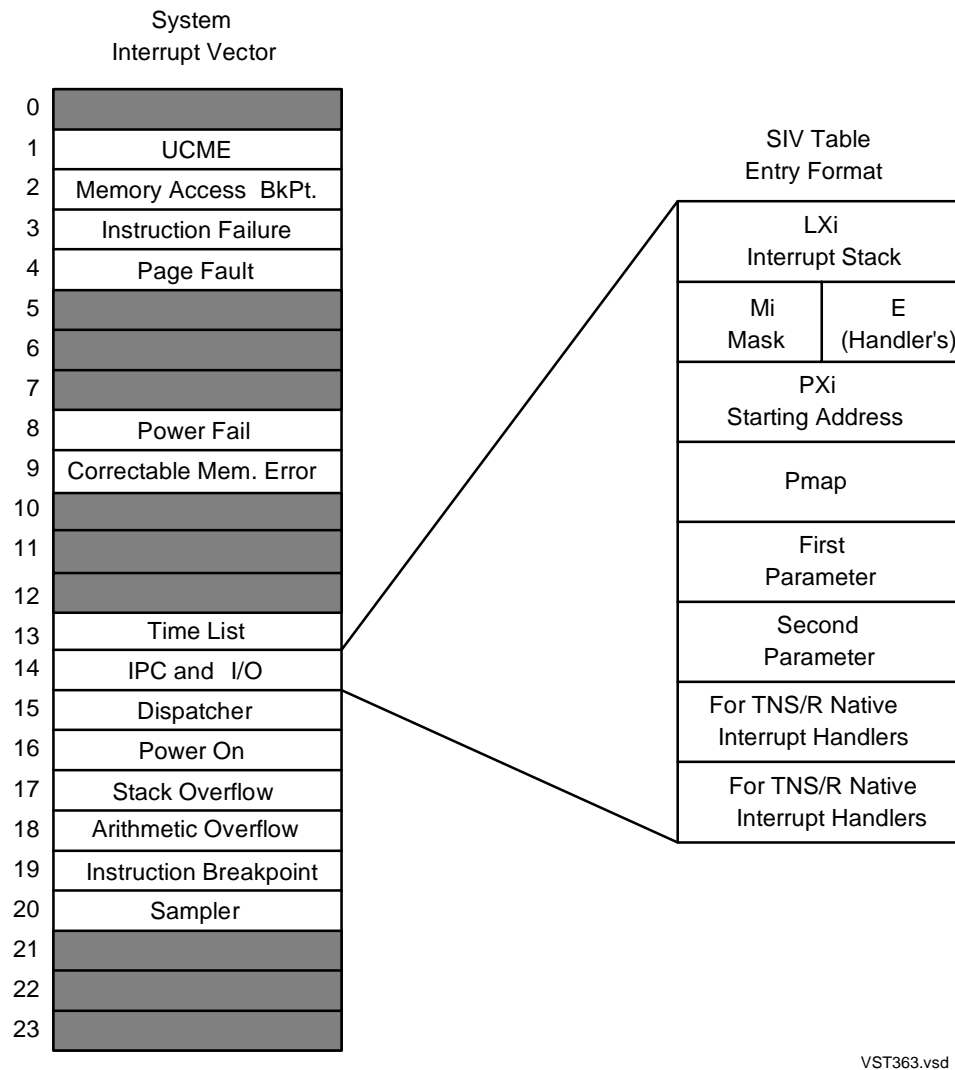
At this point, the first instruction of the interrupt handler executes. The interrupt handler runs to completion unless the interrupt handler's current mask allows certain interrupts and those interrupts do occur.

Finally, the interrupt handler executes an IXA (interrupt exit) instruction sequence, which restores the interrupted environment saved in the interrupt stack marker. That is, all registers, state values, and the program counter are returned to their preinterrupt values. Also, the register stack (if it was necessary to save it) is restored.

After the IXA instruction sequence finishes, the restored version of the Mask A register determines the next action. If no interrupt is pending, process execution (or some other interrupt handler, or the idle process) resumes at the point of interruption. If

another interrupt is pending, the interrupt sequence for that interrupt begins, using its own SIV entry to set up the interrupt environment.

**Figure 8-4. The System Interrupt Vector Transfers Control to Software**



# Interrupt Masking

Four TNS registers, simulated by exception-handling millicode, are associated with interrupts: two 16-bit interrupt registers (INTA and INTB) and two 16-bit **mask registers** (Mask A and Mask B). (Mask A is frequently referred to as “Mask” because it is the only one recognized by the operating system.) The bit assignments of these registers are illustrated in [Figure 8-5](#). In general, the priority of interrupts is in top-to-bottom order as shown in the figure—that is, uncorrectable memory error is the highest and dispatcher the lowest. However, certain interrupts are exceptions to this general rule and may be preemptive or mutually exclusive with other interrupts.

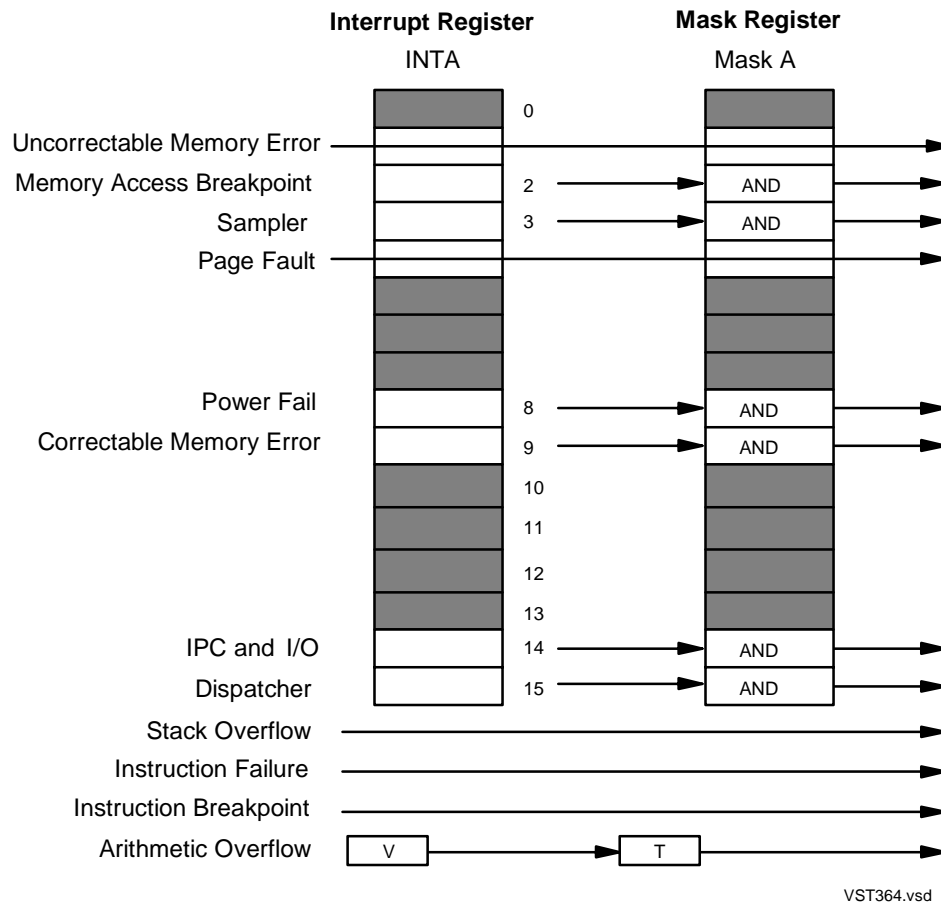
Only certain TNS interrupts are maskable—that is, the corresponding bits of the associated mask register are used by the operating system to allow or defer particular interrupt types at various times. The checking operation is performed by a logical AND of the interrupt and mask registers. The interrupt condition is ignored if the corresponding mask bit is equal to 0, and it will continue to be deferred until the mask bit is set to 1.

Bits 1 and 4 of Mask A are not used as masks, although they are assigned. Thus when the defined interrupt condition occurs, the interrupt handler is entered regardless of the state of the mask bit. Accordingly, setting the corresponding interrupt register bit would serve no purpose and so these bits also are never set (although assigned). A partial exception is INTA.<1> and its corresponding Mask A bit (uncorrectable memory error), which millicode does use in some circumstances.

The V bit (overflow bit in the Environment register) substitutes for bit 6 of INTA (formerly assigned as arithmetic overflow or divide by zero), and it is separately masked by the trap enable bit (T) of the Environment register to enable or disable that interrupt. The V bit behaves differently from other interrupt bits; it gets cleared or set by the most recent instruction that can affect it. (Other interrupt bits persist until explicitly cleared by the exception millicode or the interrupt handler.)

If two or more interrupt conditions exist simultaneously (assuming a maskable condition has its corresponding mask register bit set), the interrupt type with the highest priority (lowest bit number) takes precedence. The others are deferred until the higher-priority interrupts have been processed.

Interrupts for stack overflow, instruction failure, and instruction breakpoint have entries neither in the interrupt registers nor in the mask registers; these cause an interrupt whenever they occur, ignoring priority. The page fault interrupt, although it does have an (unused) assignment in INTA and Mask A, also causes an interrupt whenever it occurs, ignoring priority.

**Figure 8-5. Some Interrupts Can Be Masked by Mask Register Bits**

# TNS Interrupts

The following paragraphs describe the TNS interrupts in order of their SIV numbers. Each type requires zero, one, or two parameters. [Table 8-1](#) on page 8-14 summarizes the interrupts.

Uncorrectable Memory Error (1)	This interrupt occurs when a memory word is accessed by the processor and contains an error that cannot be corrected. Two parameter words are placed in the SIV entry. The first parameter word contains the physical address of the page at fault. The second parameter word identifies the type of access (data or instruction) and provides the syndrome bits generated by the error correction circuitry. The syndrome bits provide information for service providers.
Memory Access Breakpoint (2)	This interrupt occurs when the memory breakpoint has been armed by the SMBP instruction and the breakpoint memory address has been accessed in the desired manner. There is no parameter. No interrupt occurs if the breakpoint was armed by the service processor (SP); in this case, the processor performs a system freeze and enters the halt loop. This interrupt operates with less spatial precision and timing precision than on TNS processors.
Instruction Failure (3)	This interrupt occurs when an unimplemented instruction is executed, or when execution of a privileged instruction is attempted by a program that is not in privileged mode, or when an abnormal condition is detected during the execution of other instructions. The first parameter is not used. The second parameter contains a trap code that distinguishes among the various kinds of instruction failures, and an indication of how the error was detected.
Page Fault (4)	This interrupt occurs when an attempt is made to access an absent memory page (that is, its page table entry “valid” bit is set to 0). The first parameter word is the absolute byte address of the absent page.
Power Fail (8)	This interrupt occurs when a processor power failure is detected. The interrupt handler provides an orderly shutdown sequence that executes before power drops to an inoperative level. There is no parameter.
Correctable Memory Error (9)	This interrupt occurs when a memory error occurred and can be corrected. The parameter words are of the same form as those for an uncorrectable memory error.



Time List (13)	Every 10 milliseconds, the millicode detects an interval clock interrupt, updates the quadrupleword clock at SG[%350], and decrements the wait time of the element at the head of the time list. If the wait time has gone to zero, control passes to the time list interrupt handler; otherwise, no action is taken. There is no parameter.
IPC and I/O (14)	This interrupt occurs when a ServerNet interrupt requires servicing. There is no parameter.
Dispatcher (15)	This interrupt occurs when a DISP or SNDQ instruction is executed, when a process-time timeout occurs, or when a PSEM or VSEM instruction that requires operating system aid is executed. The first parameter identifies which of these events caused the interrupt.
Power On (16)	This interrupt occurs when power is applied following a power failure when memory is in a valid state. There is no parameter for this interrupt.
Stack Overflow (17)	This interrupt occurs when S exceeds 32,767 (the limit of the memory stack) following the execution of any TNS instruction (or Accelerator-generated equivalent) that can increase the S register setting (PCAL, XCAL, DPCL, ADDS, BSUB, or PUSH) or set it (SETS). In accelerated mode, the interrupt is less precise. There is no parameter.
Arithmetic Overflow (18)	This interrupt occurs when the T (trap enable) and V (arithmetic overflow) bits in the ENV register are simultaneously set to 1. The first parameter is not used. The second parameter contains a code indicating the specifics of the occurrence. In accelerated mode, this interrupt is less precise.
Instruction Breakpoint (19)	This interrupt occurs when one of several possible RISC BREAK instructions occur. This can happen when a TNS BPT (or its accelerated equivalent) instruction is executed. The first parameter is a code that identifies the kind of instruction breakpoint that occurred.
Sampler (20)	If the Measure system performance monitor is enabled, a sampler interrupt is generated at a pseudorandom interval. The interrupt handler is entered unconditionally. There is no parameter.

**Table 8-1. System Interrupt Vector Entries**

<b>SIV Number</b>	<b>Description</b>	<b>Number of Parameters</b>
1	Uncorrectable memory error	2
2	Memory access breakpoint	0
3	Instruction failure	1
4	Page fault	1
8	Power fail	0
9	Correctable memory error	2
13	Time list	0
14	IPC and I/O	0
15	Dispatcher	1
16	Power on	0
17	Stack overflow	0
18	Arithmetic overflow or divide by zero	0
19	Instruction breakpoint	1
20	Sampler	0

# 9

# Interprocessor Communication

This section describes the hardware, software, and operating sequences by which processors in NonStop S-series servers communicate with each other. Such communication is termed **interprocessor communication** (IPC) and is achieved by the exchange of **messages** between processes.

To provide transparency as to whether two communicating processes are in the same or different processors, all message traffic is managed by the **message system**. The message system uses the IPC mechanisms described in this section when the two processes reside in different processors. The message system itself, which consists of a library of message-system procedures and driver procedures that are part of the NonStop Kernel operating system, is not extensively described in this section or this manual.

Before reading this section, you should have read [Section 2, Principles of System Operation](#), for a basic understanding of ServerNet transactions and ServerNet packets. ServerNet transactions (request and response) form the basis for all communication through the ServerNet fabrics.

The topics covered in this section are:

[Interprocessor Protocols](#)

[Linker-Listener Protocol](#)

[Message System Protocol](#)

[Message Transfer Mechanisms](#)

[Message Transfer Methods](#)

[Request With Short Request Data](#)

[Request With Medium Request Data](#)

[Request With Long Request Data](#)

[Reply and Reply Acknowledge](#)

# Interprocessor Protocols

[Section 2, Principles of System Operation](#), showed that ServerNet devices, including those associated with processors, communicate with each other by exchanging ServerNet packets and that each single exchange is a ServerNet transaction. The originating device sends a read or write request packet, and the receiving device sends back a corresponding read or write response packet. In the particular case of a processor device, locally originated transactions are handled by the processor's block transfer engine (BTE); remotely originated transactions are handled by the processor's access validation and translation (AVT) logic. Both the BTE and the AVT use direct memory access to **push** or **pull** data between memory buffers and the ServerNet hardware.

[Figure 9-1](#) shows that interprocessor communication actually involves communication between at least four distinct peer layers, with each layer using its own protocol.

At the lowest level, as described in the first paragraph, the BTE and AVT logic in the processors communicate through the ServerNet hardware using the ServerNet protocol that each request packet requires a corresponding response packet. Buffer addresses are supplied by higher layers and are made available through memory tables.

At the next higher level, the message system drivers in the two processors communicate with each other using the message transfer protocol, consisting of the **pre-push protocol** and the **post-pull protocol**. The message system drivers in the two processors determine which of these two protocols to use depending on the length and direction (read or write) of the data that needs to be sent, as well as the protocol demands of the higher layers (such as needing to transmit their own kind of acknowledgments). Having made that determination, ServerNet services then direct the BTE and AVT hardware to carry out the message transfer, one packet at a time.

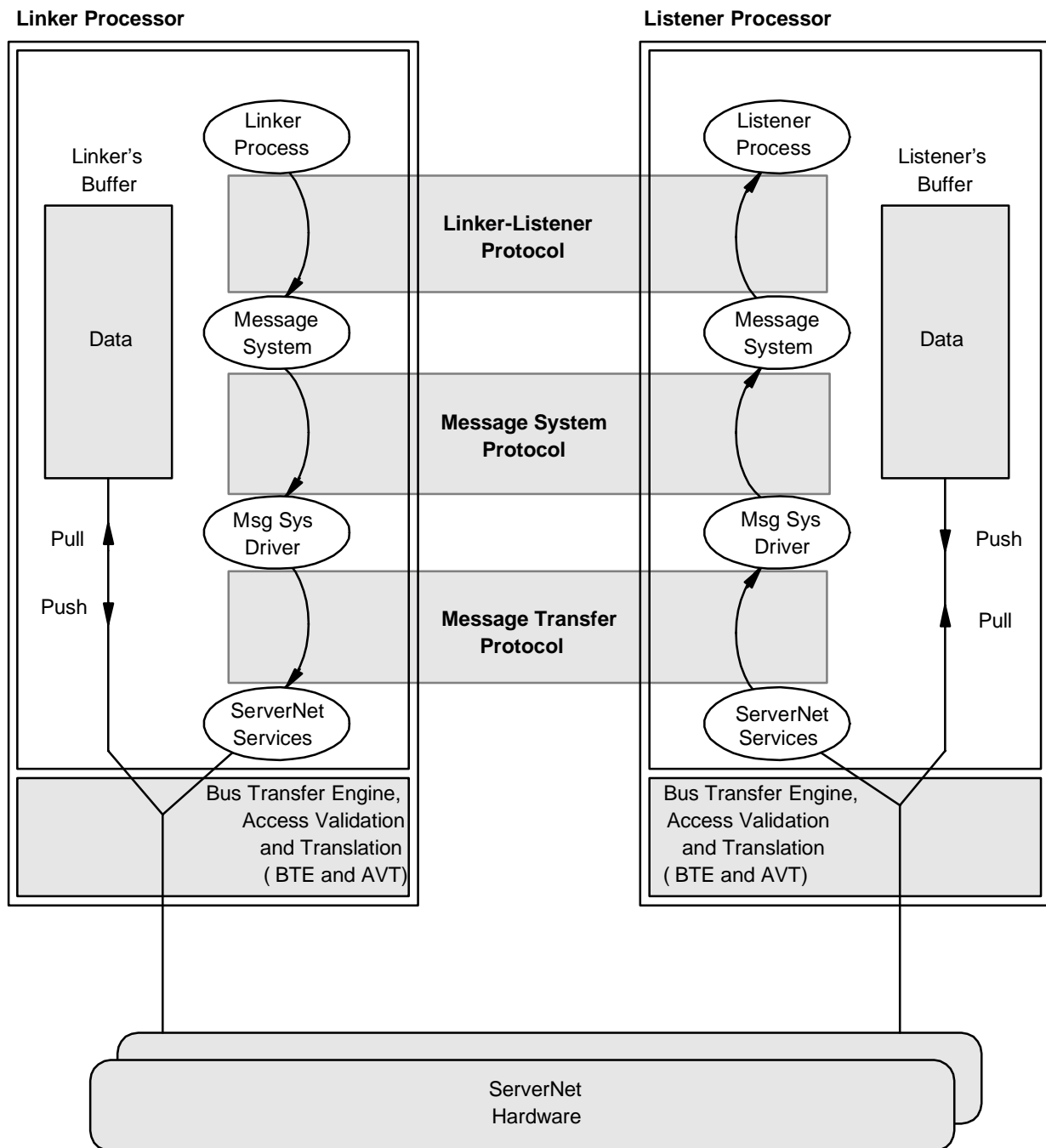
At the next higher level, the message systems in the two processors communicate with each other by exchanging setup information and additionally an acknowledgment for each message sent. Setup information, sent in a **setup packet**, includes such things as the byte count of the data and the ServerNet addresses of where the message information is to be written or from where it is to be read. The message information includes control information passed to the message system from its client (the next-higher layer). From all this information (control, data, and setup), the message system creates messages for transmission by the ServerNet services. If the combination of control information and data exceeds a certain length (approximately 61 kilobytes, currently), the message systems create multiple messages as necessary.

At the highest level (at least as shown in [Figure 9-1](#)) are the two processes that originated the need for a particular interprocessor communication. The **linker process** invokes the message system to deliver a particular request, and the **listener process** receives the request from its message system and returns a reply after it has fulfilled the request. The linker and listener might be some kind of application program interface (API) executing on behalf of some other application or user process, or they

might be the file system and thus not a process at all. The processes shown are generic representations for all such cases.

For convenience of reference throughout this section, the processor containing the linker process is called the linker processor and the processor containing the listener process is called the listener processor.

**Figure 9-1. Interprocessor Communication Involves Multiple Levels of Protocol**



VST338.vsd

# Linker-Listener Protocol

[Figure 9-2](#) separately shows the linker-listener protocol as if lower levels of protocol did not exist. However, it is those lower levels that actually carry out the actions shown here.

The linker-listener protocol consists of a request (1) and a reply (2). Optionally, data can be included in either the request or the reply—or both or neither, depending on what the linker process wants to do. Data included in the request is called request data, and data included in the reply is called reply data.

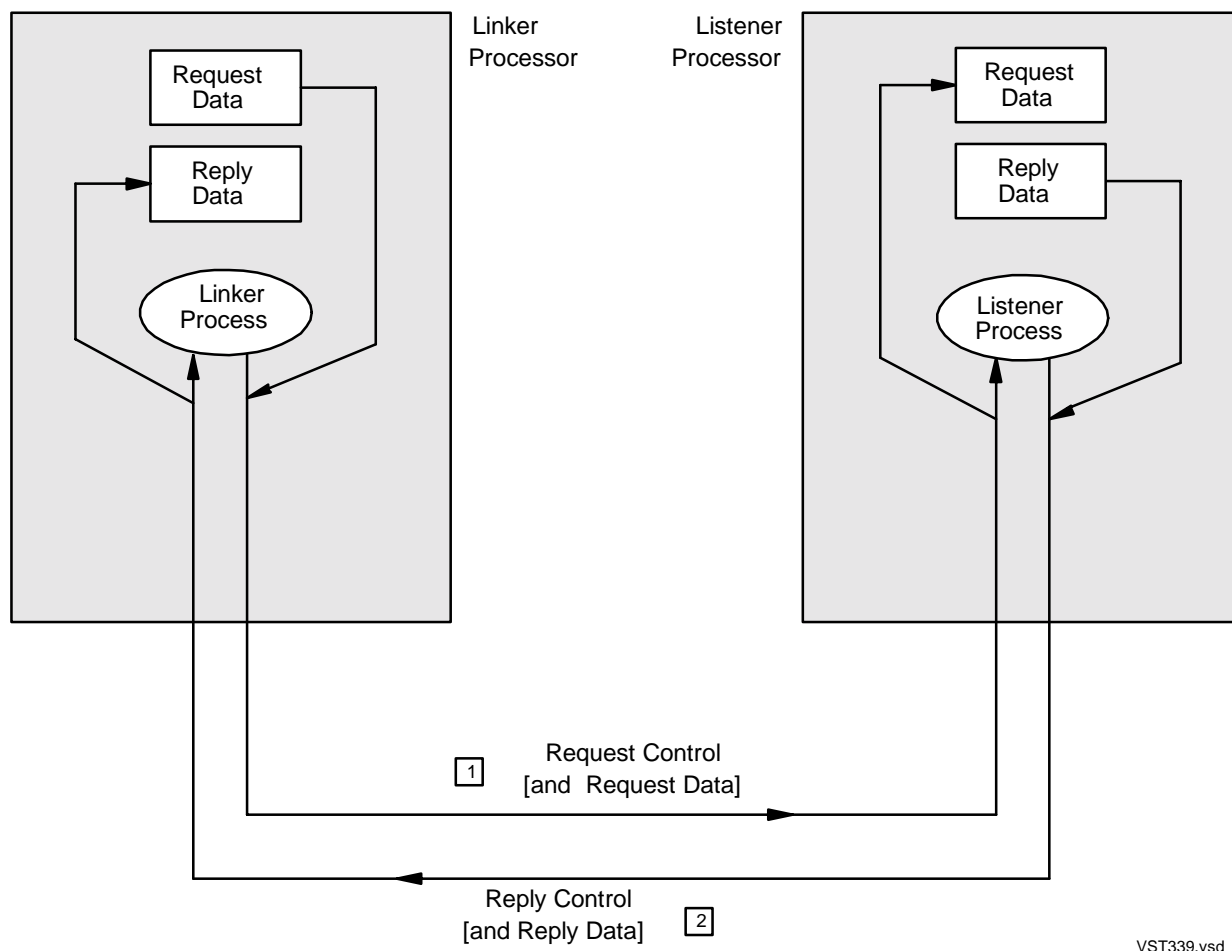
Control information is always included in a message. This control information is stored in buffers that are physically separate from the data buffers and are called request control buffers and reply control buffers. For simplicity, however, [Figure 9-2](#) combines control and data storage, and considers all these buffers to be “data” buffers.

The linker process begins the communication by invoking the message system, instructing it to send request control information (and optionally data, such as for a write request) to the listener process. The message-system call used by the linker to create and send the message is `MSG_LINK_`.

After the listener processor has received the message, the listener processor immediately stores the control information (and request data, if any) either directly in the listener's request data buffer (as shown) or in temporary space in a cache or small buffer—depending on the combined size of the control and data. Then the message system alerts the listener process that a message has arrived. Accordingly, the listener process calls its message system with `MSG_LISTEN_` to find out basic information about the message: its size, message identification, whether request data was included, and so on. With that information, the listener again calls the message system with `MSG_READCONTROL_` to obtain the control information, which provides processing requirements, such as reading or writing data. If request data was stored in a cache, the listener also calls `MSG_READDATA_` to transfer the message data into its request data buffer.

After the request has been fully handled, the listener again calls its message system (with `MSG_REPLY_`) to send a reply message back to the linker process. The reply message always contains control information, providing status about the status of the request. This informs the linker that the data has (or conceivably has not) been written. If reply data is to be included in the reply, such as for a read request, the listener includes such data in its reply.

When the linker processor receives the reply message, it immediately stores the reply data in the linker's reply data buffer, and the message system alerts the linker process that a message has arrived. Accordingly, the linker uses `MSG_BREAK_` to obtain the reply control information from the message system, which provides status information about the listener's handling of the original request. This call also informs the linker that reply data, if any, is present in the reply data buffer.

**Figure 9-2. From Linker-Listener Perspective: One Request and One Reply**

# Message System Protocol

When the linker and listener processes exchange messages back and forth (as described in the preceding topic), the message system adds its own protocol information to those same messages. In [Figure 9-3](#), that added information is shown enclosed in shaded boxes in the messages; the unshaded information in the messages is exactly the same as shown in [Figure 9-2](#). Details about the movement of control and data between processors, described in the preceding topic, are therefore omitted in this description. Note, however, that message system icons have been added and the flow of control information to and from the message systems is shown.

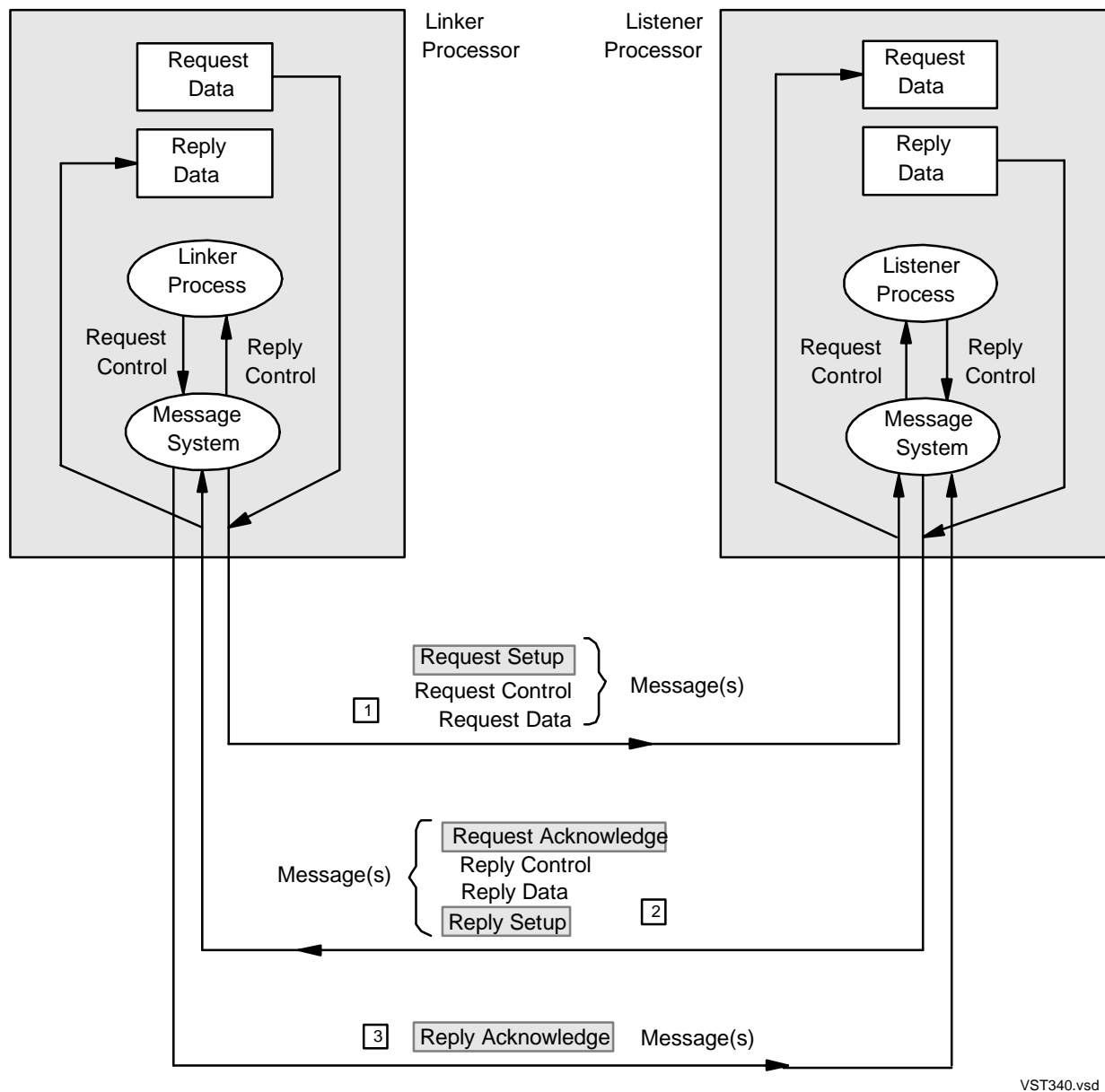
In comparison with [Figure 9-2](#), a primary difference in the messages is that request setup information has been added to the request messages and reply setup has been added to the reply messages. As mentioned earlier, setup information tells the peer message system what the byte count of the data is (if present) and provides ServerNet addresses for the various buffers.

In addition to exchanging setup information, it also is the responsibility of the message systems to send an acknowledgment back to the sending message system whenever a request message or a reply message has been received. Such acknowledgments can, for the sake of reducing message traffic, be included in (“piggy-backed” on) any setup packet going in the same direction to the same processor.

If there happens to be no setup packet in which the acknowledge can be included, a one-packet message needs to be sent back separately to the other processor’s message system after the message has been received. That is the case shown as an example in [Figure 9-3](#), wherein the reply acknowledge is returned separately and alone from the linker to the listener in both cases. The reply acknowledge could have been returned in a request setup going in the same direction.



**Figure 9-3. The Message System Uses Setup and Acknowledgments in Messages**



# Message Transfer Mechanisms

Referring back to [Figure 9-1](#) on page 9-3, which showed the four levels of protocol that are involved in interprocessor communication, three have at this point been described. The ServerNet hardware protocol was covered in [Section 2, Principles of System Operation](#), and the linker-listener and message system protocols were covered in the preceding two topics. The remainder of this section describes the fourth, remaining protocol. This topic and the next one provide some basic information, and the four final topics provide specific sequences for transferring messages through the network.

[Figure 9-4](#) shows the mechanisms involved in message transfers between processors. Later illustrations in this section use simplified representations of the details shown here. Refer back to this figure whenever you need clarifying detail.

Whenever a client process (which could be either a linker or a listener) needs to send a message, it uses the mechanisms denoted by the numbers 1 through 6 in the figure. The client (1) is responsible for allocating some buffer space for the communication. (In addition to these buffers, the message system has setup buffers that it maintains in its own setup areas.) Then the client uses the linker-listener protocol to invoke the message system (2) to begin the message transfer.

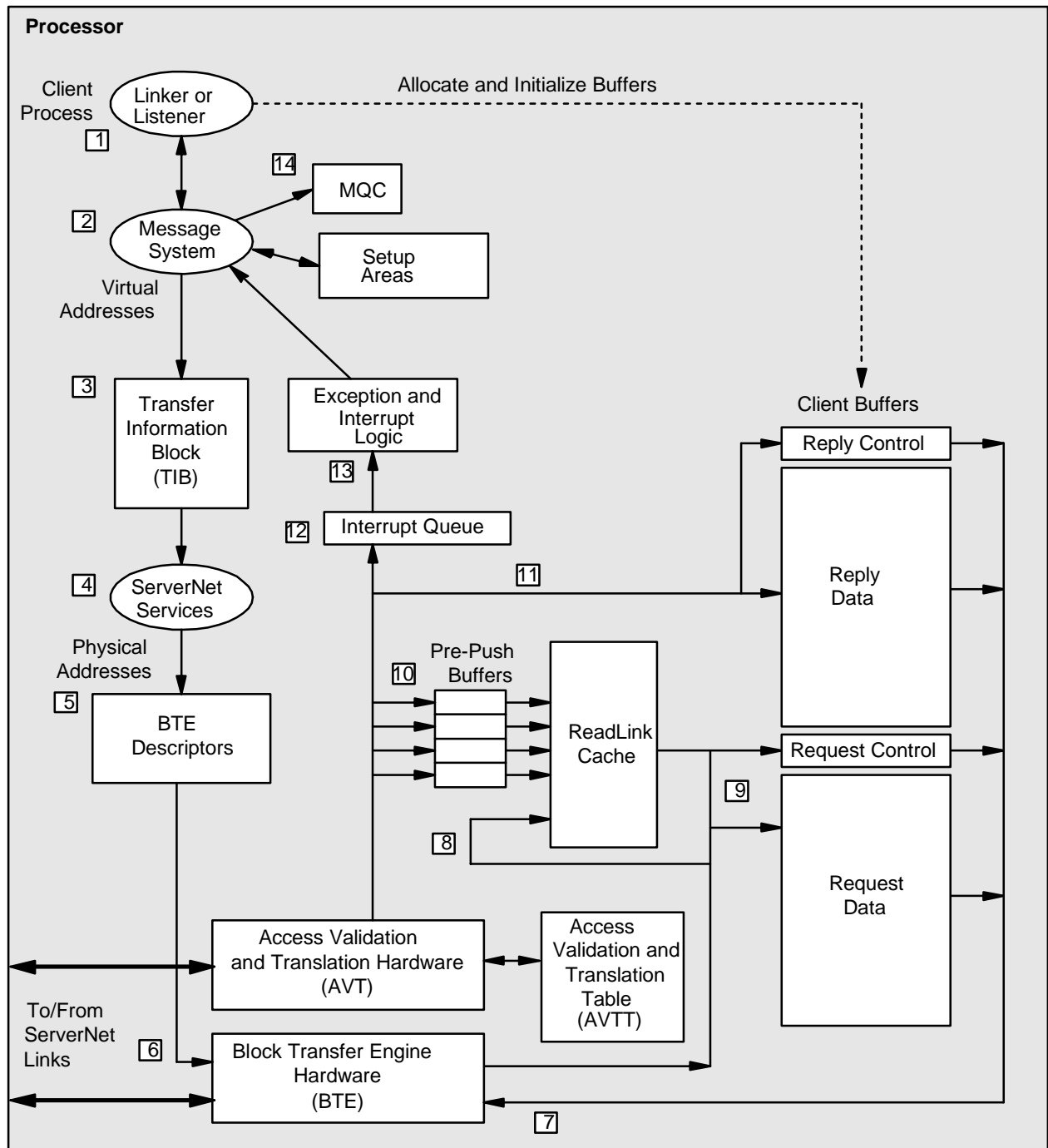
The message system creates a transfer information block (TIB) (3), and calls ServerNet services (4) to assume control. ServerNet services translates the virtual addresses in the TIB to physical addresses as a linked list of BTE descriptors (5). Then ServerNet services passes control to the hardware of the block transfer engine, or BTE (6). The BTE manages (or at least starts to manage) the physical transfer of information to and from the client buffers, transmitting and receiving such information over the ServerNet hardware to and from the buffers of the corresponding process in the target processor. However, for some kinds of transfers (as shown later), the target processor could switch roles, so that it actually performs the pulling or pushing of data with its own BTE.

Depending on the kind of transfer and particular phase of the transfer, the BTE can either push data from the buffers (7) or pull data into the buffers (8 and 9). Pushing data involves either request control and request data, or reply control and reply data. Pulling of data by the BTE is for request control and data only, and follows one of the two paths shown, depending on the combined size of the control and data. If the combined size will fit into the readlink cache (which does not require address translation), the control and data are stored there (8) for later copying to the client buffers. Otherwise, the data is stored directly into the addressed buffers (9).

When the processor receives a ServerNet transmission that originated at another node, that transmission is handled by the access validation and translation (AVT) hardware. The AVT stores the data wherever specified in the ServerNet packets. Short messages, such as those consisting only of control information, are usually addressed to and stored in one of four pre-push buffers (10). Longer messages, such as reply information with combined control and data size exceeding 512 bytes (current specification), are addressed to and stored in the client's reply buffers (11).

When the remote processor sends an interrupt packet (such as to signify the end of a transfer), the packet is placed in the interrupt queue (12) for processing by different levels of exception and interrupt logic (13), and the message system stores the information in an MQC (14).

**Figure 9-4. Message Data Is Transferred To and From Request and Reply Buffers**



VST341.vsd

# Message Transfer Methods

[Message System Protocol](#) on page 9-6 shows that different kinds of transfers can use different combinations of mechanisms. Sometimes the target processor of a message can reverse roles to pull data from the originator of the message, and the request data can be handled different ways depending on the length of the data.

The upper part of [Figure 9-5](#) tabulates the three basic phases that are used as a basis for succeeding figures. Because the message system protocol requires a reply to each request, there is always a reply upon which to piggy-back the acknowledgment to a request, as shown in [Figure 9-5](#). Therefore a separate acknowledge request phase is never necessary. Frequently, also, the acknowledge reply can be piggy-backed on some other message going in the same direction. However, a piggy-backed acknowledge reply is not always possible, and so a separate acknowledge reply phase is shown.

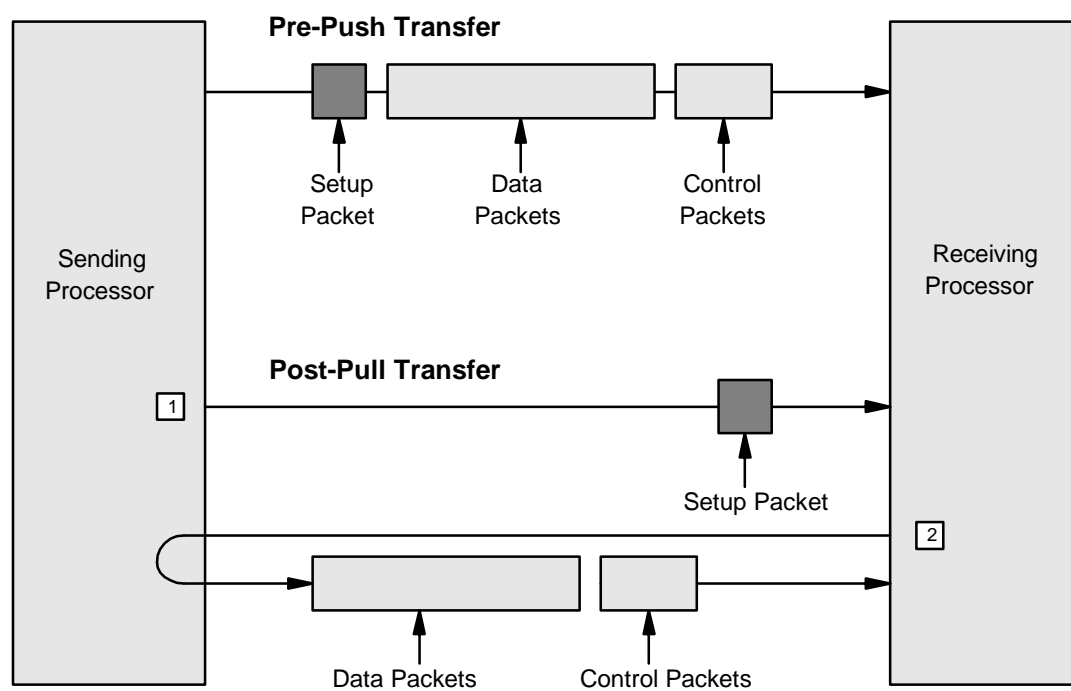
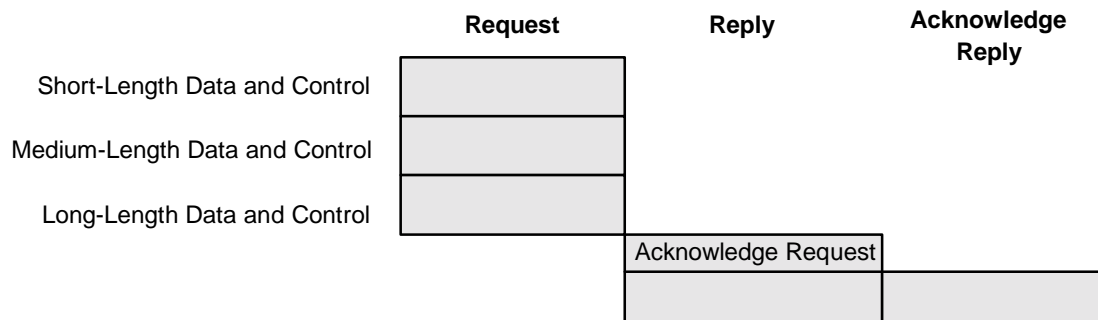
The request phase is implemented with three different mechanisms, one being chosen depending on the combined length of the control and data information in the request. The next three topics ([Request With Short Request Data](#), [Request With Medium Request Data](#), and [Request With Long Request Data](#)) separately consider each of these techniques of request processing. The final topic ([Reply and Reply Acknowledge](#)) considers the reply and reply acknowledge phases.

In different circumstances, either the request phase or the reply phase can use one of two basis transfer methods—pre-push transfer or post-pull transfer.

The lower part of [Figure 9-5](#) illustrates the difference between pre-push and post-pull transfer methods. That difference is based on the placement of the setup packet in a message. In the pre-push case, data is pushed by the sending processor from the source buffer to the destination buffer ahead of (“pre-”) the setup packet. In the post-pull case, the sending processor first pushes the setup packet (1), then the data is pulled (2) by the receiving processor from the source buffer into the destination buffer after (“post-”) the setup packet has been received. The control packets precede data in either case.

Wherever possible, the message system uses the pre-push method. Because the setup packet incurs an interrupt in the receiving processor, it is more efficient to send the data first and then interrupt the processor to notify it that data is waiting, rather than to interrupt first and then send the data packet by packet. However, the pre-push method can be used only for transfers that fit in a pre-push buffer (currently 512 bytes).

Every processor has four pre-push buffers for each processor that could be sending messages to it. Therefore, in a 16-processor system, 15 processors could send messages to any other, and there would be 15 times 4, or 60, pre-push buffers in each processor.

**Figure 9-5. Message Transfers Use Three Phases and Two Transfer Methods**

VST342.vsd

# Request With Short Request Data

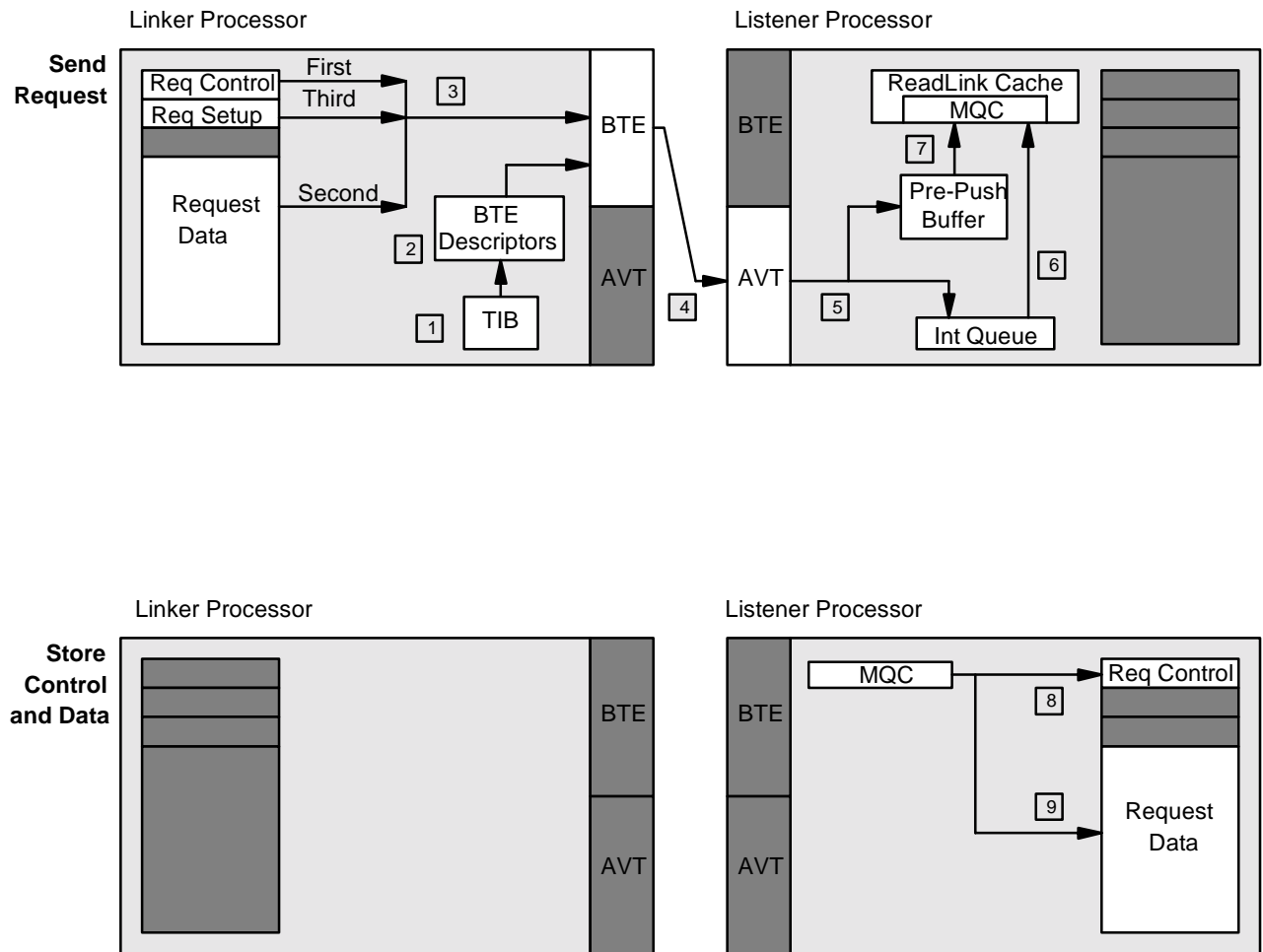
A request phase that includes short-length request data ([Figure 9-6](#)) makes temporary use of pre-push buffers to transfer data from a linker's buffer to a listener's buffer. This transfer method is used when the total length of control information and data does not exceed the size of a pre-push buffer, currently 512 bytes. Because the addresses of these buffers are static and known to the message systems in each processor, the transfer can proceed without the linker knowing the ultimate location of the listener's destination buffer.

In the first part of this phase (upper part of [Figure 9-6](#)), all of the request information and the request data is sent from the linker to the listener as a single message. For this message, the message system creates a TIB (1), using the ServerNet address of one of the four pre-push buffers known to exist in the listener's processor specially for this (the linker's) processor. ServerNet services translates the TIB to BTE descriptors (2), and transfers control to the BTE.

The BTE (3) pre-pushes the request control information first, then all of the data, and lastly transmits the request setup packet. Each packet is transmitted over the ServerNet hardware (4) to the AVT hardware in the listener processor, where it is sent to one of two destinations (5). The control information and the data are sent to the addressed pre-push buffer and the final packet, the setup, is linked into the interrupt queue.

When the interrupt is processed, the message system transfers the setup information to an MQC in the readlink cache (6), as well as the control information and all the data (7). The message system then alerts the listener process that the message has been received.

In the second part of this phase (lower part of [Figure 9-6](#)), the listener process calls the message system with MSG\_READCONTROL\_ and MSG\_READDATA\_ to copy the control information and data in the MQC in the readlink cache to corresponding buffers (8 and 9). The listener process might have allocated these buffers previously or could do so at this point.

**Figure 9-6. For Request With Short Data, Listener Pre-Pushes Data**

VST344.vsd

## Request With Medium Request Data

When the combination of request control and data is too big to fit in the pre-push buffers but small enough to fit in a readlink cache buffer ([Figure 9-7](#)), the message system eliminates use of the pre-push buffers (shown in [Figure 9-6](#) on page 9-13). Instead, the message system causes the listener process to reverse roles and to **pull** the data to itself, using ServerNet response packets to transfer data from a linker's buffer to the listener's readlink cache. (In the two preceding topics, data was sent in ServerNet request packets—that is, pushed.)

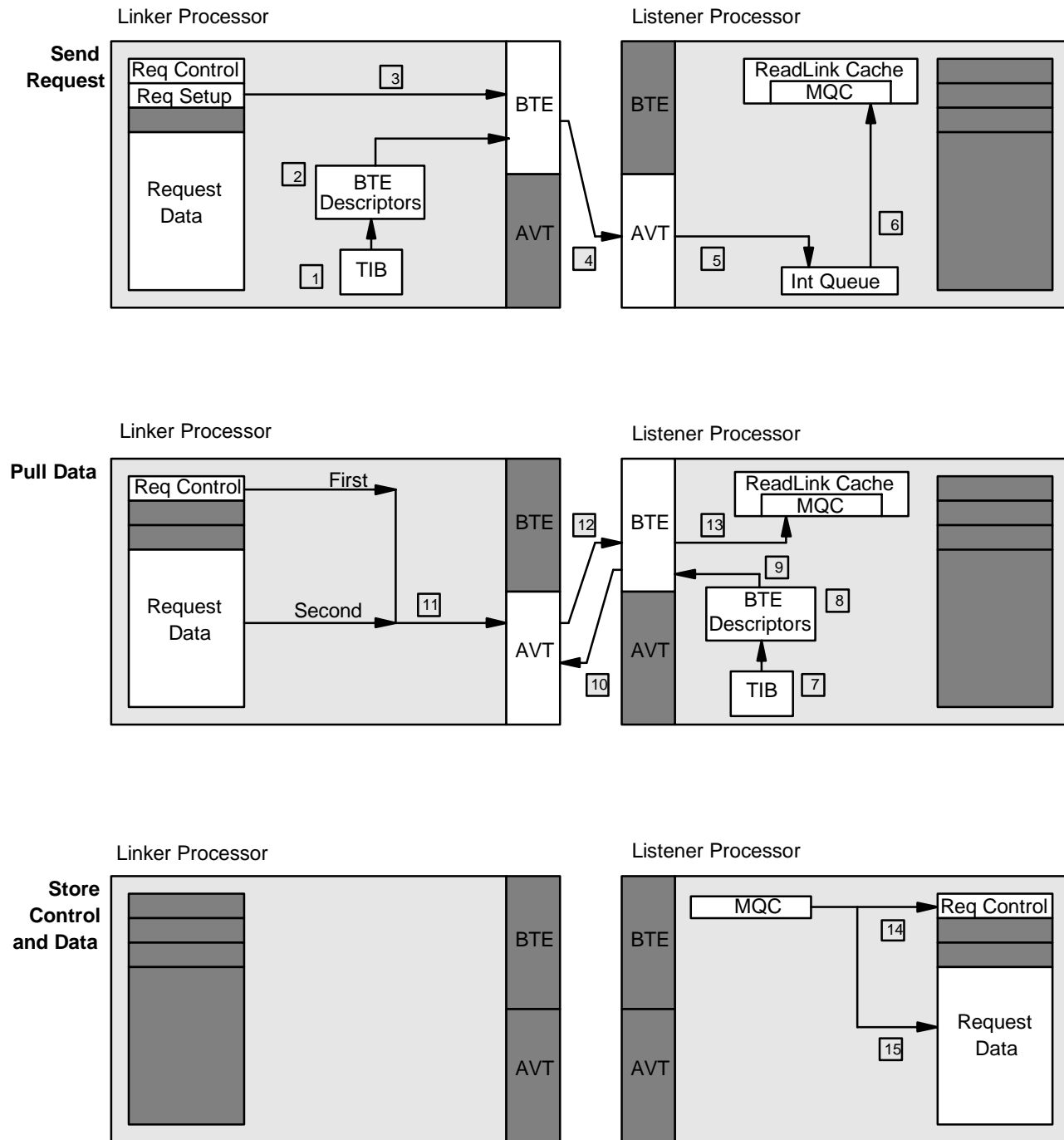
In the first part of this phase (upper part of [Figure 9-7](#)), only the setup information is sent from the linker to the listener. This is done as a single-packet message. For this message, the message system creates a TIB (1), and ServerNet services translates the TIB to BTE descriptors (2). ServerNet services transfers control to the BTE (3), which transmits the request setup packet (4) to the AVT hardware in the listener processor. There, the setup is linked into the interrupt queue (5).

When the interrupt is processed, the message system reads the setup information, stores it in an MQC in the readlink cache (6), and recognizes that this request contains request data of medium length. It consequently recognizes that it has to pull the request data.

In the next part of this phase (middle part of [Figure 9-7](#)), the listener's message system creates a TIB (7) in preparation for the pull. ServerNet services then translates the TIB to BTE descriptors (8) and transfers control to the BTE (9). The BTE in the listener then proceeds to issue read request packets to the AVT in the linker processor (10), which in corresponding response packets, transmits the original request control information, followed by the request data (11). The response packets go back to the listener BTE (12) for temporary storage in the readlink cache (13).

In the third part of this phase (lower part of [Figure 9-7](#)), the listener process calls the message system with MSG\_READCONTROL\_ and MSG\_READDATA\_ to copy the control information and data in the MQC in the readlink cache to corresponding buffers (14 and 15).



**Figure 9-7. For Request With Medium Data, Listener Post-Pulls Data to Cache**

VST345.vsd

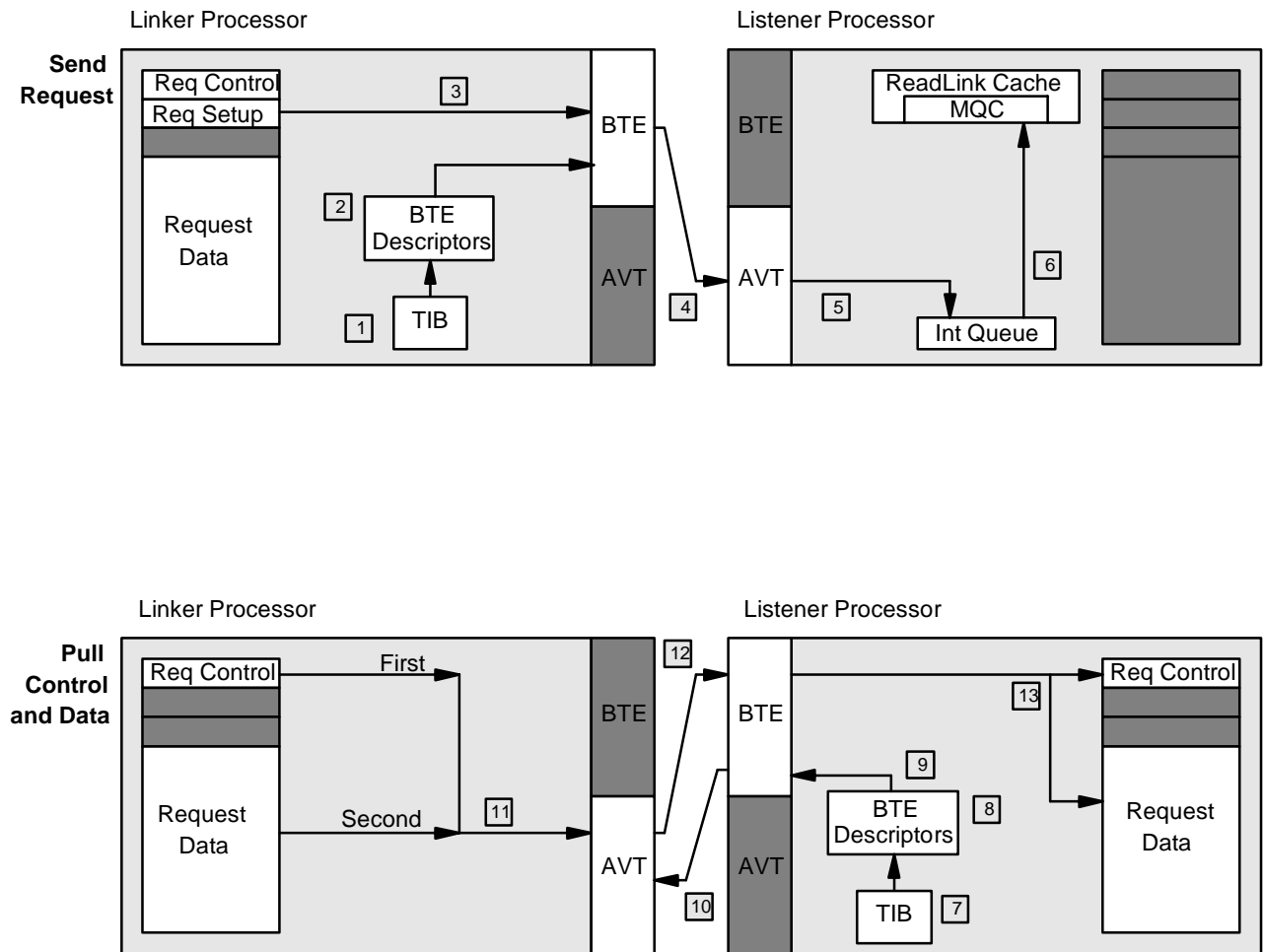
# Request With Long Request Data

When the combination of request control and data is too big to fit in either the pre-push buffers or the readlink cache, the message system eliminates use of both of these elements and instead pulls the data directly to the listener's buffer. This operation is shown in [Figure 9-8](#).

In the first part of this phase (upper part of [Figure 9-8](#)), only the setup information is sent from the linker to the listener. This transfer is done as a single-packet message. For this message, the message system creates a TIB (1), and ServerNet services translate the TIB to BTE descriptors (2). ServerNet services transfers control to the BTE (3), which transmits the request setup packet (4) to the AVT hardware in the listener processor. There, the setup is linked into the interrupt queue (5).

When the interrupt is processed, the message system reads the setup information, stores it in an MQC in the readlink cache (6), and recognizes that this request contains request data of long length. Because the message system cannot handle this request by itself at this level, it goes to the listener and notifies it that it needs to pull the request data directly to the request data buffer.

In the second part of this phase (middle part of [Figure 9-8](#)), the listener successively calls MSG\_READCONTROL\_ and MSG\_READDATA\_ to pull the request control information and the request data to itself. For each of these calls, the listener's message system creates a TIB (7). ServerNet services translates the TIB to BTE descriptors (8) and transfers control to the BTE (9). The BTE in the listener then proceeds to issue read request packets to the AVT in the linker processor (10), which, in corresponding response packets, transmits the original request control information, followed by the request data (11). The response packets go back to the listener BTE (12) for storage in the client buffers of the listener process (13).

**Figure 9-8. For Request With Long Data, Listener Post-Pulls Data to Its Buffer**

VST346.vsd

# Reply and Reply Acknowledge

Every time a listener receives a request, whether short, medium, or long, it must respond with a reply. That response occurs in a reply phase, which may or may not necessarily include data. But whether or not reply data is included with the reply, the linker, after it receives the reply, must respond with an acknowledgment.

Such a reply acknowledgment can be piggy-backed on some other message going to the listener processor but, if not, a separate acknowledge reply transmission is required—thus both events are shown as separate phases in [Figure 9-9](#).

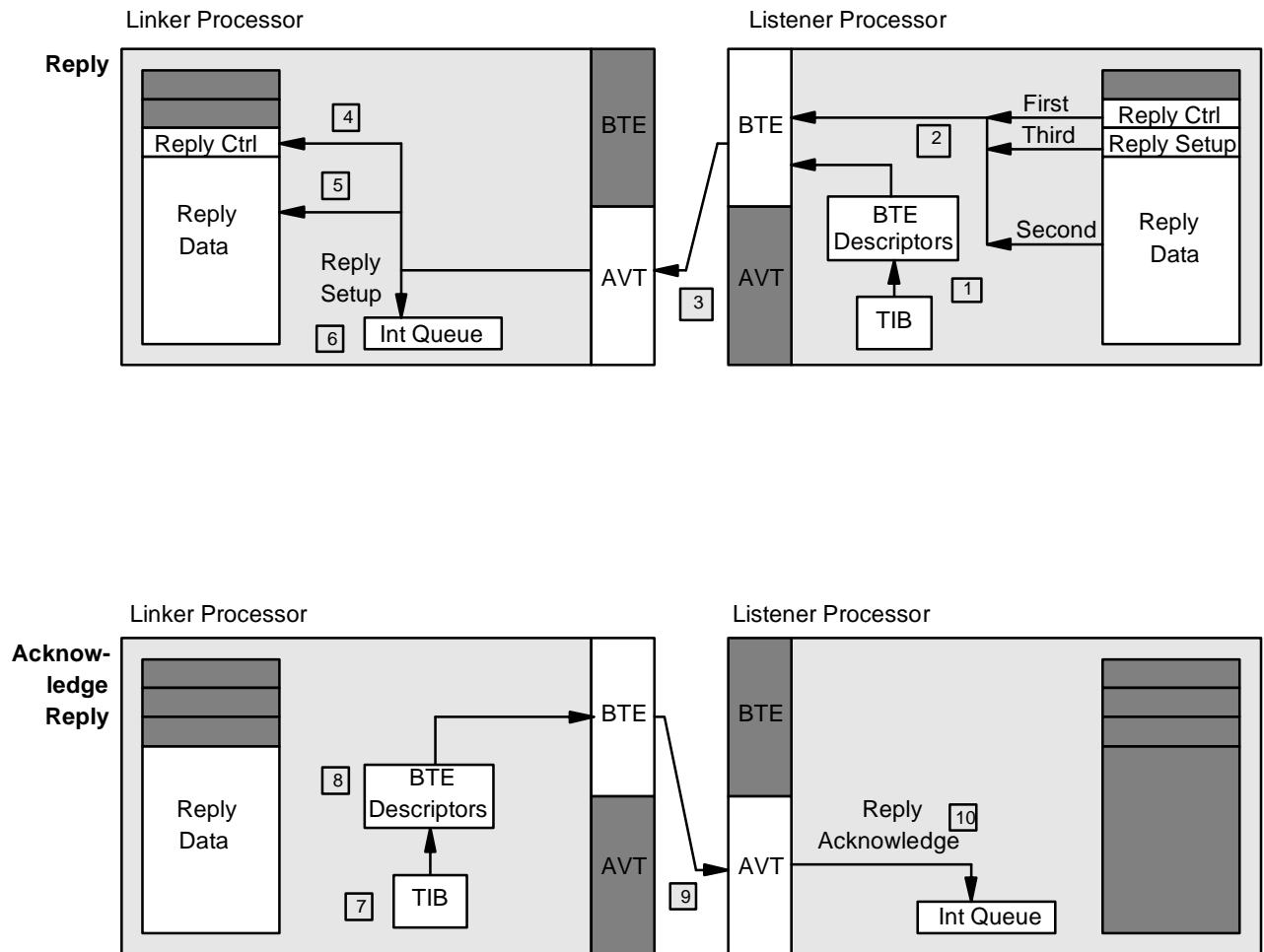
Whenever a reply contains reply data, such as a reply to a read request, the reply phase transfers data from a listener's buffer to a linker's buffer. Because the address of the linker's buffer is known to both the linker and the listener, it is not necessary to use pre-push buffers for delivery of the reply data; data is transferred directly from one client buffer to the other without intermediate copying.

The sequence of events, as illustrated in [Figure 9-9](#), is as follows.

In the reply phase (where the request requires reply data), the listener process has already read the request control information and now knows that it needs to reply with the appropriate reply data. It creates its own control information in its reply control buffer and calls the message system with MSG\_REPLY\_. The message system accordingly creates a new TIB, which ServerNet services translates to BTE descriptors (1).

When the BTE hardware transmits the reply message (2), it pre-pushes the reply control packets, then all the data packets, and finally the setup packet. (This setup packet, or possibly some prior setup packet, includes the request acknowledge information.) These packets are received in order by the linker's AVT hardware (3) and stored according to their different contents. The reply control packets (4) and the reply data packets (5) are stored in their respective linker buffers. The reply setup packet, the last one to arrive, is sent to the interrupt queue (6), so that the message system, when interrupted, can alert the linker process that the reply data has been received.

In the acknowledge reply phase, the linker process calls its message system with a MSG\_BREAK\_ call to send a final reply back to the listener (acknowledging receipt of the reply message) and to retire the original request message. If possible (often the case), the message system will include (piggy-back) the acknowledge-reply information in some other message going to the same destination. Otherwise, the message system builds a simple TIB (7), which gets translated to a BTE descriptor (8), and a resulting interrupt packet is sent through the ServerNet hardware (9) to the listener's interrupt queue (10). The message system on the listener side, when interrupted, then alerts the listener process that the reply message has been received by the linker, and resources used by this read request can be freed.

**Figure 9-9. For All Read Requests, Listener Pre-Pushes Data in Reply Phase**

VST343.vsd



# 10 Input/Output Operations

This section describes the operating sequences by which processors in NonStop S-series servers transfer data to and from input/output devices. Before reading this section, you should have read [Section 2, Principles of System Operation](#), for a basic understanding of ServerNet transactions and ServerNet packets. ServerNet transactions (request and response) form the basis for all input and output operations through the ServerNet fabrics.

Doing communications input/output is so fundamentally different from doing storage input/output that the operations need to be described separately. The first three topics in this section provide a comparison of similarities and differences, and the remainder of the section separately discusses *storage* operations (next five topics) and then *communications* operations (final five topics).

Topics in this section are:

[Storage and Communications I/O Compared](#)

[Overview of the I/O System](#)

[Layers of I/O Components](#)

[I/O Process Models for Storage I/O](#)

[Storage Operation Queuing](#)

[Packaging and Delivery of Storage Commands](#)

[Storage Read Request Processing](#)

[Storage Write Request Processing](#)

[Communications Operation Queuing](#)

[Application of Communications Queues](#)

[Typical Use of Queue Pairs](#)

[Actions for Empty or Full Queues](#)

[Communications Request Processing](#)

# Storage and Communications I/O Compared

[Figure 10-1](#) illustrates the basic similarities and differences between storage I/O and communications I/O. Both modes of operation transfer information through the ServerNet hardware whenever data needs to be exchanged between a processor and a controller. However, workstations on an external network conceivably could communicate among themselves for long periods of time without needing ServerNet transfers, whereas, in the case of storage I/O, all data transfers are through the ServerNet hardware.

In the case of storage I/O, there is generally an application of some kind that originates the need for an I/O transfer. It sets up a buffer for receiving information from a storage device, or sets up and fills a buffer for transmission to a storage device. Then the application, through other software that typically includes the file system, sends its I/O transfer request to an I/O process. The I/O process converts the request to commands appropriate for the I/O device (such as SCSI commands) and sends these commands through the ServerNet hardware to the controller associated with the target device.

At that point, still in the case of storage I/O, the controller assumes control of the I/O operation, issuing commands to the I/O device and interacting with the ServerNet hardware to transfer the specified data to or from the client's data buffer. Transfers on both sides of the ServerNet hardware, to and from the controller and to and from the processor, are achieved with direct memory access (DMA), requiring no involvement of the I/O process.

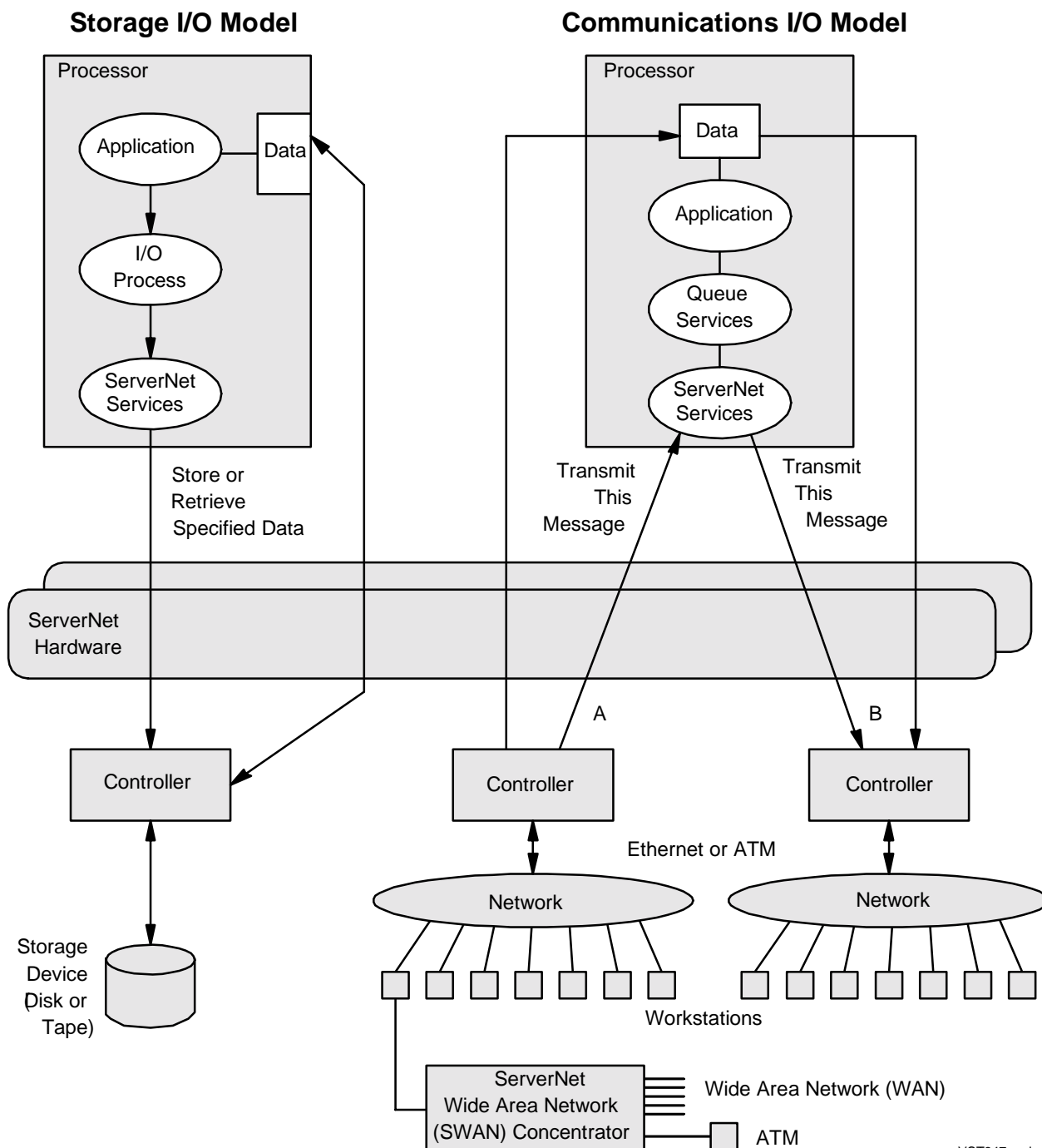
In the case of communications I/O, there is no I/O process. Instead, the communication is between a NonStop host application in the server and an application in one of several workstations. Transfers that are inbound to the host server (A) originate externally by some request from a workstation or other device that is part of an external network. Transfers that are outbound from the host server (B) typically provide information or control from some service in the NonStop processor.

Queue services in the processor maintains an orderly list of such requests in circular queues, so that sudden bursts of requests involving some particular resource can be accommodated fairly. When a particular request rises to the top of its queue, queue services and a controller interact through ServerNet services to transfer the data using DMA on both sides of the ServerNet hardware.

Supported communications protocols include local area networks (LANs) and Asynchronous Transfer Mode (ATM). As shown, wide area networks (WANs) and ATM circuits interface to NonStop servers through a ServerNet wide area network (SWAN) concentrator. A Common Communications ServerNet adapter (CCSA) is available to which support the SS7 protocol.



**Figure 10-1. Application Initiates Storage I/O, Communications I/O Uses Ethernet or ATM**



# Overview of the I/O System

[Figure 10-2](#) expands upon information presented in [Section 1, Introduction](#) (see [Figure 1-8](#) on page 1-15). In this figure, those components that make up the ServerNet hardware are highlighted in the central part of the illustration. These details are mostly eliminated in succeeding illustrations for the sake of simplicity.

Starting at the top, note that the processor typically has several layers of I/O software; the actual components in these layers differs between storage I/O and communications I/O, as explained in the previous topic. In either case, however, both kinds of I/O interface to the ServerNet hardware through a collection of procedures called ServerNet services. The ServerNet services provide the hardware-accessible tables, buffers, and descriptors that the processor ServerNet interface can use for transferring ServerNet packets through the ServerNet links. The primary components of the processor ServerNet interface are the bus transfer engine (BTE) and the access validation and translation (AVT) logic.

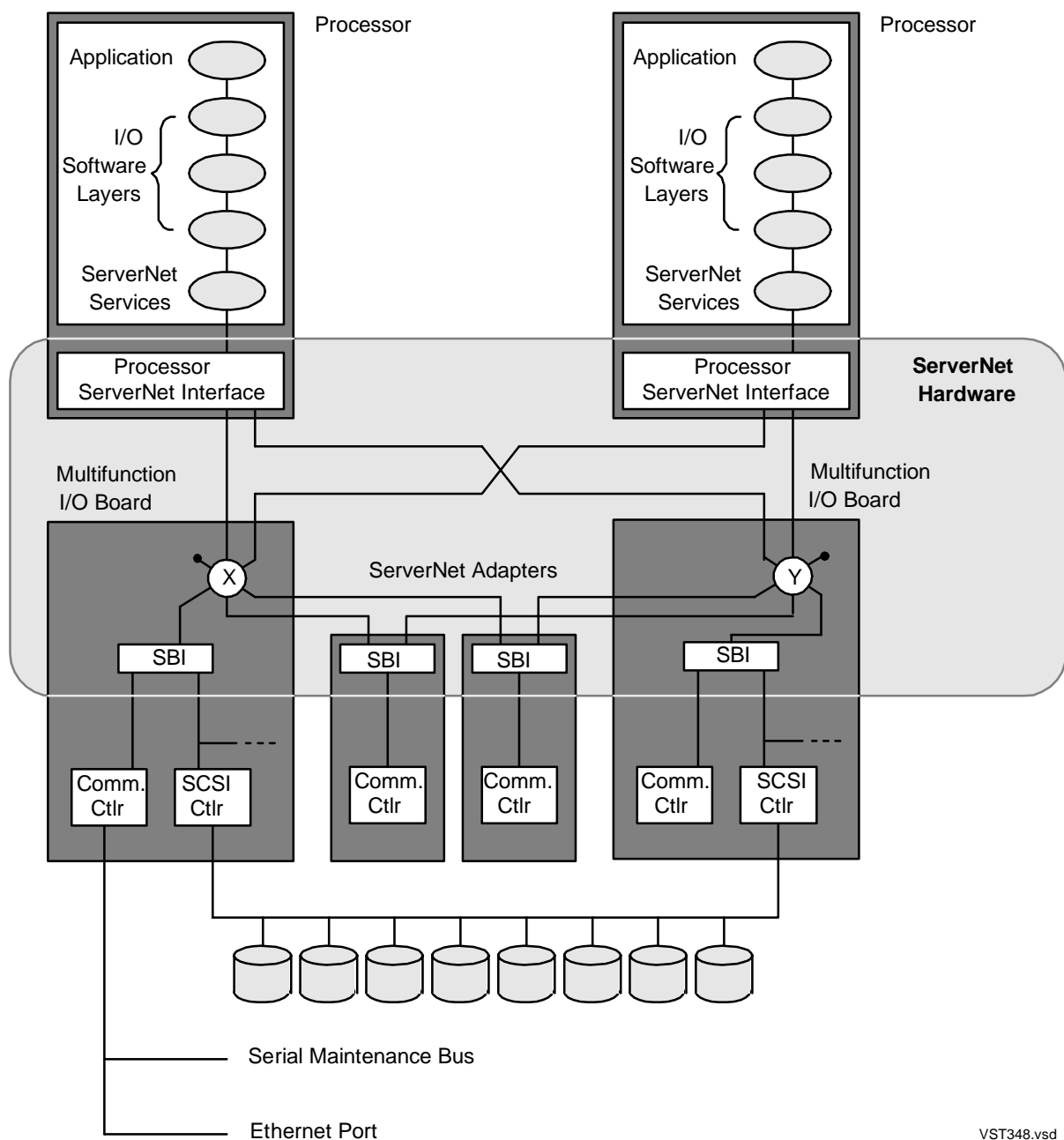
ServerNet routers (two of which are shown, marked X and Y) provide massive expandability of the ServerNet links, as detailed in [Section 1, Introduction](#). On the multifunction I/O boards (MFIOBs) within either a processor enclosure or an I/O enclosure, these routers provide ServerNet links to a ServerNet bus interface (SBI) on the same MFIOB, and two ServerNet adapters. The ServerNet adapters, unlike the MFIOBs, connect to both X and Y fabrics. The unused router ports in the figure are available for ServerNet expansion.

The ServerNet bus interfaces provide conversion of ServerNet protocol to whatever protocol is used by the buses associated with the various controllers. Depending on the kind of bus that is used, a secondary converter may be required such as in the case of the SCSI protocol. A single SBI can, depending on design, provide bus conversion for several controllers—four in the case of the MFIOB.

Each controller has its own ServerNet ID, as explained in [Section 2, Principles of System Operation](#) (see [Figure 2-2](#) on page 2-5), and is therefore fully described as a ServerNet addressable controller (SAC). Each controller has its own RISC processor and local memory, from which it runs its own native code and specially modified code for ServerNet usage.

There are three SCSI controllers on the MFIOB. Two of these controllers are matched in pairs with the two MFIOBs in an enclosure, so that each SCSI bus can be controlled through either the X fabric or the Y fabric. These two buses are typically used for multiple disk drives. The third SCSI port, a differential SCSI port, is typically used for a tape drive.

The communications controller on the MFIOB provides a port for the serial maintenance bus (SMB) and an Ethernet port.

**Figure 10-2. ServerNet Hardware Bridges Controllers to I/O Software**

VST348.vsd

# Layers of I/O Components

[Figure 10-3](#) illustrates the various software and hardware levels of I/O control that accomplish input/output transfers, with storage I/O illustrated on the left and communications I/O illustrated on the right. The most significant difference between these two types of I/O is the location of the request queues. For storage I/O, the queues are located on the ServerNet adapter board or the multifunction I/O board (MFIOB); note the boxes labeled “Global Request Queues” and, in controller memory, “Request Queues.” For communications I/O, the queues are instead located in processor memory; note the box labeled “NIOC Queues.”

Despite the difference in location of request queues, the actual transfer of data is always under control of the controller, whether for storage I/O or communications I/O. The reason for assigning data transfer responsibility to the controller is to unburden the processors from the routine of transferring data to and from the I/O devices, which have the potential to be numerous.

The top layer in the I/O system is identified as the “Managing Process.” For storage I/O this is the I/O process; for communications I/O it is a monitor process. Under the standard I/O model (see next topic), there is one I/O process (or process pair) for each logical device that can be addressed. As for monitor processes, however, there is one process (or process pair) for each generic type of communications (LAN, ATM, and so on). All processes at this level are initially started and configured by an **interface monitor** process (one process pair in a processor), which is an agent of the **subsystem manager** process (one process pair in the system). In addition, the interface monitor initializes and configures all XIO module drivers in its processor.

The next layer down consists of the “Extensible I/O (XIO) Procedures,” which include the XIO kernel and all the XIO module drivers. The **XIO kernel** routes requests and interrupts between the various module drivers and the I/O processes and monitors. It is also responsible for setting up workspaces for the module drivers. A **module driver** is a specialized set of procedures that deals directly with ServerNet services (in the case of storage I/O) or with queue services (in the case of communications I/O). It is the module driver that converts generalized I/O requests to specific commands for driving the particular hardware devices under its control.

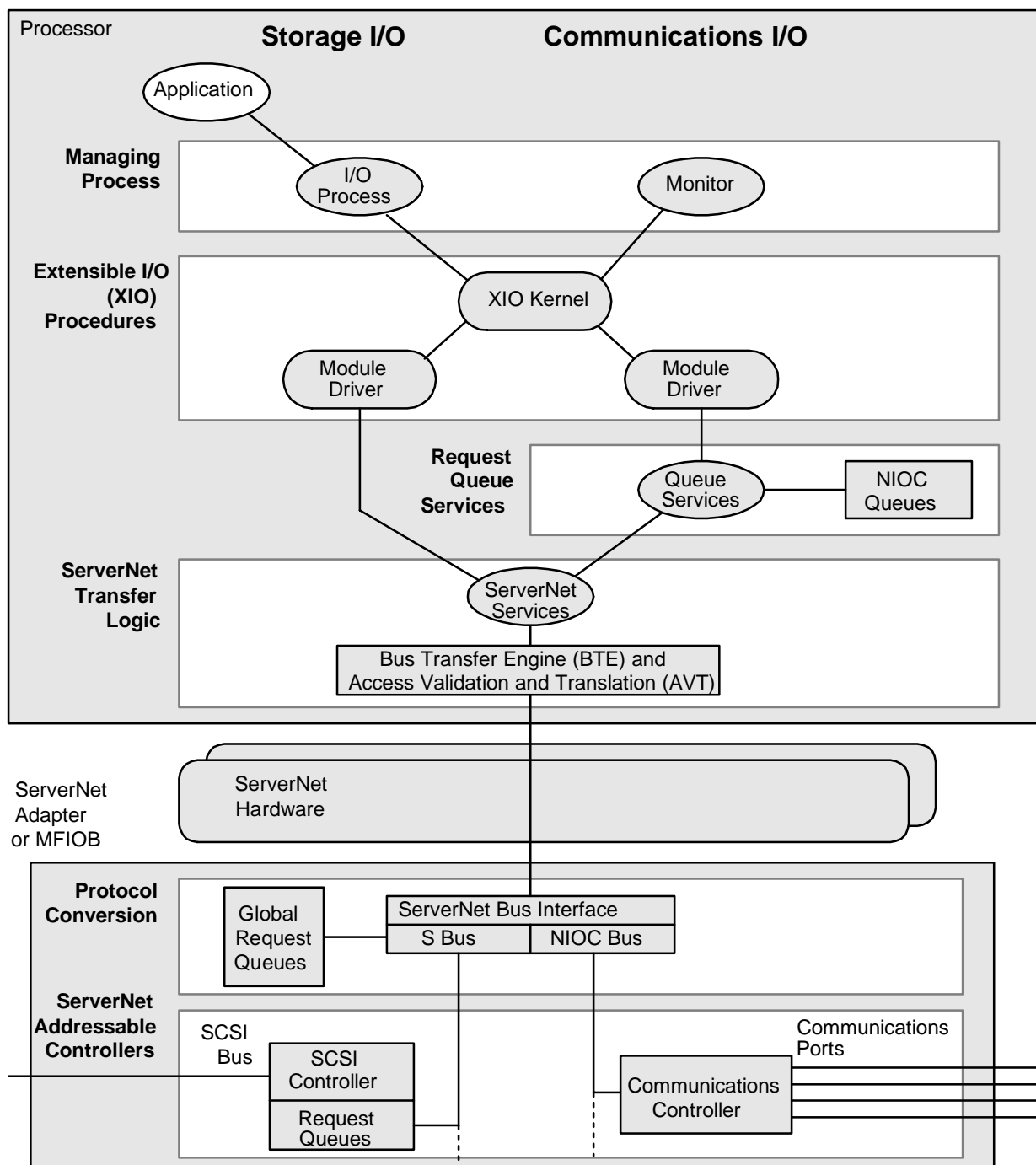
The “Request Queue Services” layer consists of the **queue services** set of procedures, which manages the request queues for communications I/O. These queues are identified as native I/O controller (NIOC) queues for the NIOC protocol used by the communications controllers.

The “ServerNet Transfer Logic” comprises both software (ServerNet services) and hardware (BTE and AVT), which together perform all processor transfers across the ServerNet hardware.

The “Protocol Conversion” layer converts ServerNet traffic to and from the form of protocol required by the particular bus to which a given controller is attached.

The “ServerNet Addressable Controllers” layer is the lowest level of the I/O architecture, carrying out the actual transfer of data across the ServerNet hardware, and providing the interface to the hardware I/O devices.

**Figure 10-3. Queues in Controllers for Storage, in Processor for Communications**



VST349.vsd

# I/O Process Models for Storage I/O

[Figure 10-4](#) shows two I/O process models. Both models use the conventional process-pair arrangement of I/O processes that is used in all NonStop servers, including those prior to the NonStop S-series servers. Those earlier servers, however, required the I/O process and its backup to be in the two processors which interfaced to the two I/O buses that connected to the relevant I/O controllers.

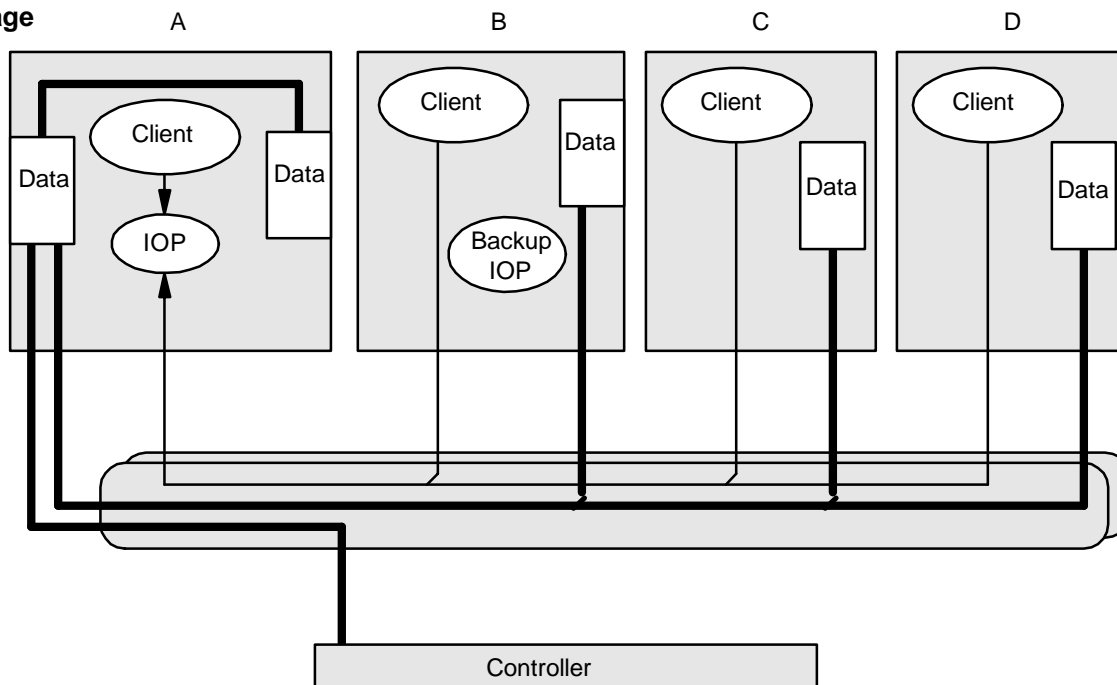
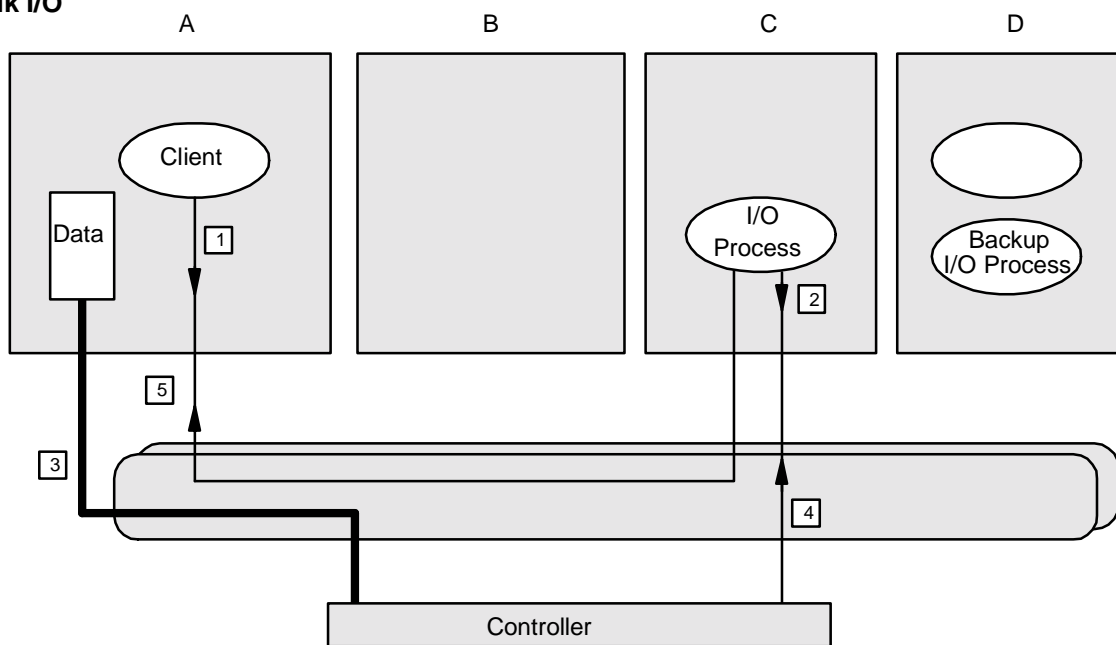
In the ServerNet case (NonStop S-series servers), the standard I/O storage model can still be used, but the restriction of locating an I/O process and its backup in adjacent processors is removed. This freedom of location happens because there are no I/O buses in the ServerNet architecture. However, for the sake of speed and efficiency, the I/O processes are usually located in processors that are in the same enclosure as the relevant I/O controllers.

In the first case (standard I/O storage model), note that the I/O process for this example is located in processor A. Its backup is located in processor B. To perform an I/O operation through this I/O process, client processes in processors B, C, and D must send their request through the message system (as described in [Section 9, Interprocessor Communication](#)) and ServerNet hardware to the I/O process in processor A (note lighter-weight lines).

Each client in this case has its own buffer space, including the client shown in processor A. However, the I/O process must transfer data to and from the controller using its own buffer (note darker line connecting to the controller), and clients must transfer the data to and from the I/O process's buffer (the other dark lines). For processors B, C, and D, this means additional transfers through the ServerNet hardware.

As an improvement on the process pair model, a direct bulk I/O transfer model (lower part of [Figure 10-4](#)) is possible (not available in G02.00 release). This model still requires each client to communicate with the primary I/O process to start the I/O transfer, but the data is transferred directly between the controller and the client's buffer without needing to go through a separate IOP data buffer. This path saves numerous ServerNet transfers.

In the direct bulk I/O example in [Figure 10-4](#), the primary I/O process is located in processor C, and its backup is in processor D. To begin an I/O transfer, the client in processor A sends a read or write request to the I/O process (1). The I/O process then initiates the transfer by sending appropriate commands to the controller (2), including the ServerNet address of the client buffer in processor A. The controller, using the supplied address, assumes control of the operation, transferring data directly between the client buffer in processor A and the storage device (3). When the transfer is completed, the controller sends a completion notice to the I/O process (4), which in turn interrupts the client to signify completion of the operation (5).

**Figure 10-4. Alternative Storage I/O Models Provide Backward Compatibility****Standard  
I/O Storage  
Model****Direct Bulk I/O  
Transfer Model**

VST350.vsd

# Storage Operation Queuing

As mentioned previously, it is the module driver that takes generalized I/O requests and formulates specific commands for driving I/O devices. [Figure 10-5](#) shows a close-up view of how such commands are queued and delivered to the controllers.

As shown in [Figure 10-5](#), **request queues** are located in two places, both on the same I/O board: either a ServerNet adapter or a multifunction I/O board (MFIOB). These two locations are identified as global and local request queues. A request queue is a circular list of 64-byte command entries (see next topic).

The **global request queues** are allocated out of global SRAM memory space from a pool of queues. Each module driver performs such allocations, using a separate and private protocol interface. The module driver organizes the queues in such a way that there is one queue for each processor that a given controller might communicate with. Because the module driver allocates these queues, it knows their ServerNet addresses when it comes time to transmit command information to any given queue. Both the module driver and the controllers maintain pointers to the head and tail of each global queue with which it is associated. (The I/O process is limited in the number of outstanding requests it can issue, so that it cannot overrun the space allocated for a queue.)

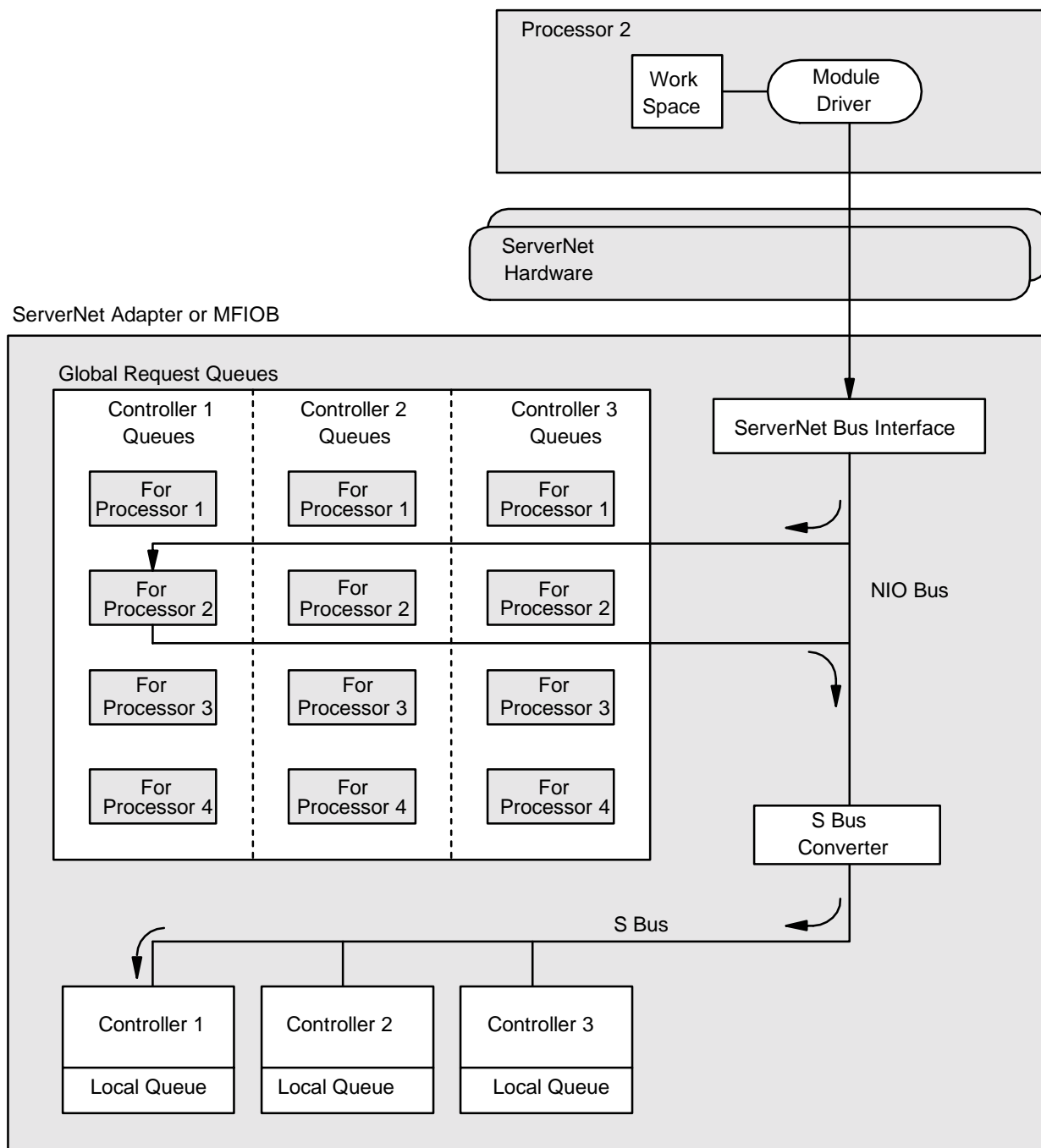
When the module driver forms a command entry, it transmits it as a single packet through the ServerNet hardware (by means of ServerNet services and the BTE, omitted for simplicity) to the ServerNet bus interface (SBI) on the adapter board. The SBI, taking the address from the packet, stores the command entry (CE) in the appropriate queue. The example in the figure assumes that the packet came from processor 2 and is destined for controller 1 on the adapter board.

Periodically, the controller 1 firmware polls its global queues and discovers that the sequence number of the newest entry differs from its own record of sequence numbers. Therefore the controller copies the entry (or entries) from the global queue to its own **local request queue** in its own local memory. In the case of SCSI storage devices, such as are on the MFIOB, the transmitted information passes through an S bus converter for electrical compatibility with S bus devices.

The controller now can operate on the command entries in its request queue. Although the controller receives the command entries in chronological order, it can choose to execute the commands in some other order.

When the I/O operation is completed, the controller sends a completion status message back to the module driver to signify successful (or other) results. This message arrives at the module driver by an indirect route. First the controller sends an interrupt packet to the interrupt queue in the processor in which the module driver is located. Later, when the interrupt is processed, the completion status is delivered to the module driver.



**Figure 10-5. Module Driver Pushes Entries to Global Buffers**

VST351.vsd

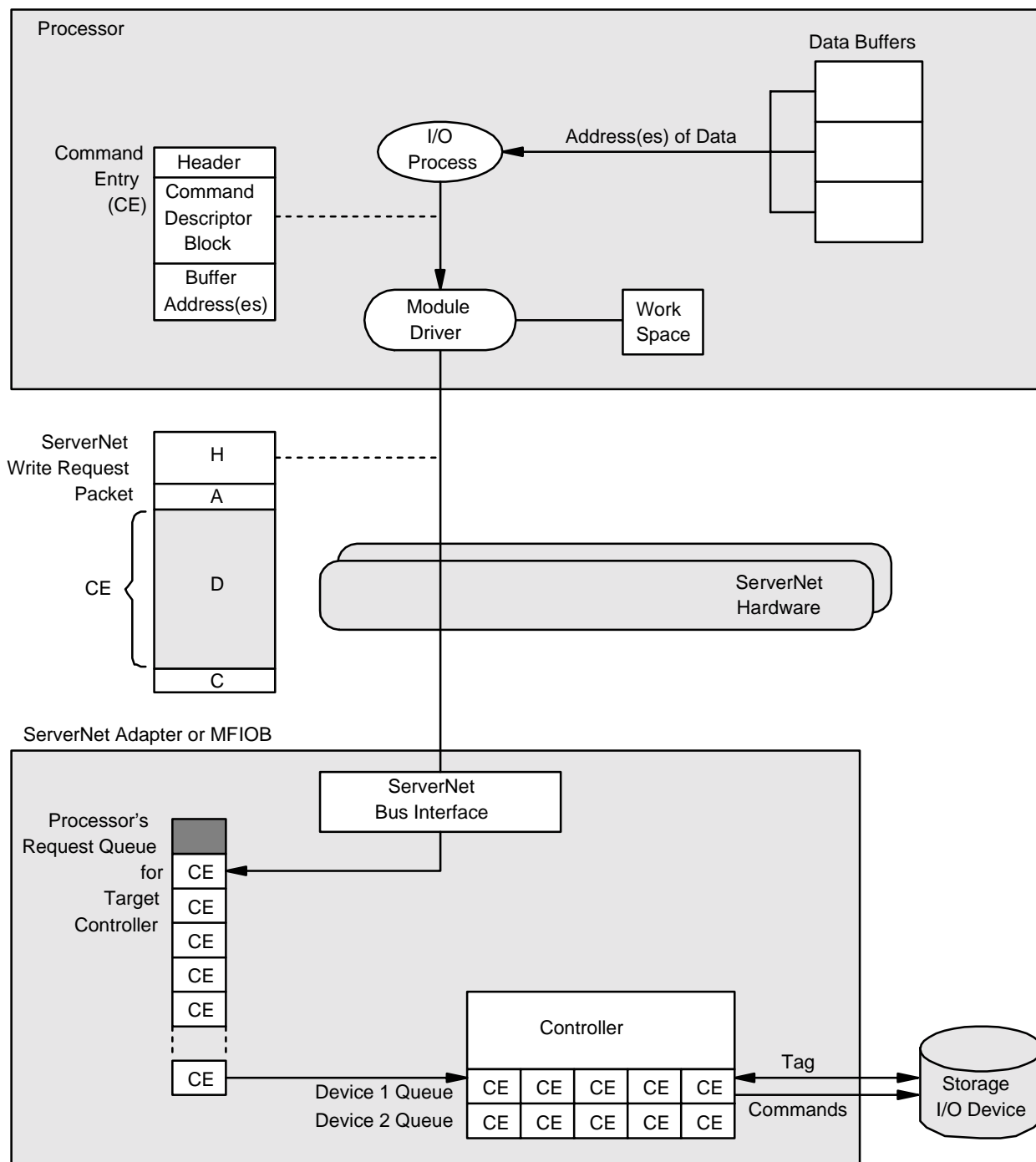
# Packaging and Delivery of Storage Commands

A **command entry** (CE) in a request queue defines an I/O operation to be executed by the controller firmware. Each such entry is 64 bytes in size, the maximum size that can be conveyed in ServerNet packets. [Figure 10-6](#) shows how these commands are transmitted to a particular storage I/O device. The example shown assumes that the device is a SCSI device.

When an I/O process needs to transfer data, it establishes one or more data buffers for reception or transmission of data. The process then creates a **command descriptor block**, which is a SCSI standard for command specification, then adds the client buffer address (or addresses) and a header to format a complete 64-byte command entry. The process then forwards the command entry to the module driver, which, by making reference to information in the CE header, obtains the ServerNet ID and ServerNet address of the destination from its work space and formats a ServerNet packet. Note that the data part of the packet (D) consists of the entire CE. The CE is preceded by the ServerNet packet header (H) and the ServerNet address (A) for writing the CE into the destination queue, and it is followed by the cyclic redundancy check code (C).

When the module driver receives this packet, it forwards the packet to ServerNet services for transmission through the ServerNet hardware to the ServerNet bus interface (SBI) that is specified by the destination ServerNet ID in the packet header. By reading the address field of the packet, the SBI is able to translate the ServerNet address to a physical address and store the CE in the correct global queue.

The controller, after it polls its global request queues, eventually transfers the CE to one of its own local queues, the one for the particular device being targeted. The device is identified by a SCSI address contained within the CE, and not by a ServerNet address or ServerNet ID. The controller uses command tags when identifying commands issued to the storage I/O device in doing the external data transfer.

**Figure 10-6. I/O Process Sends Command Descriptor Block to Controller**

VST352.vsd

# Storage Read Request Processing

When an I/O process in a processor requests a read from a storage device, the only action that the processor initiates is to send the command to the controller. Thereafter, it is the responsibility of the controller to transfer the data, pushing the data to the processor as if the controller is the originator of the request. The resultant sequence consists of three phases, illustrated in [Figure 10-7](#).

In the **request phase**, the module driver first of all assigns the data buffer a ServerNet address and makes an entry in the access validation and translation table (AVTT). Then the I/O process (1) builds a command descriptor block (CDB) and (2) embeds the CDB in a command entry (CE). The CE includes addresses of the client buffers for reception of the data that is to come later. Then the I/O process, through the XIO kernel, calls the module driver to create and send a ServerNet packet to a certain ServerNet addressable controller (3). The header of the created packet contains the destination and source ServerNet IDs, and the address field contains the ServerNet address of the next available slot in the appropriate global request buffer in the ServerNet adapter.

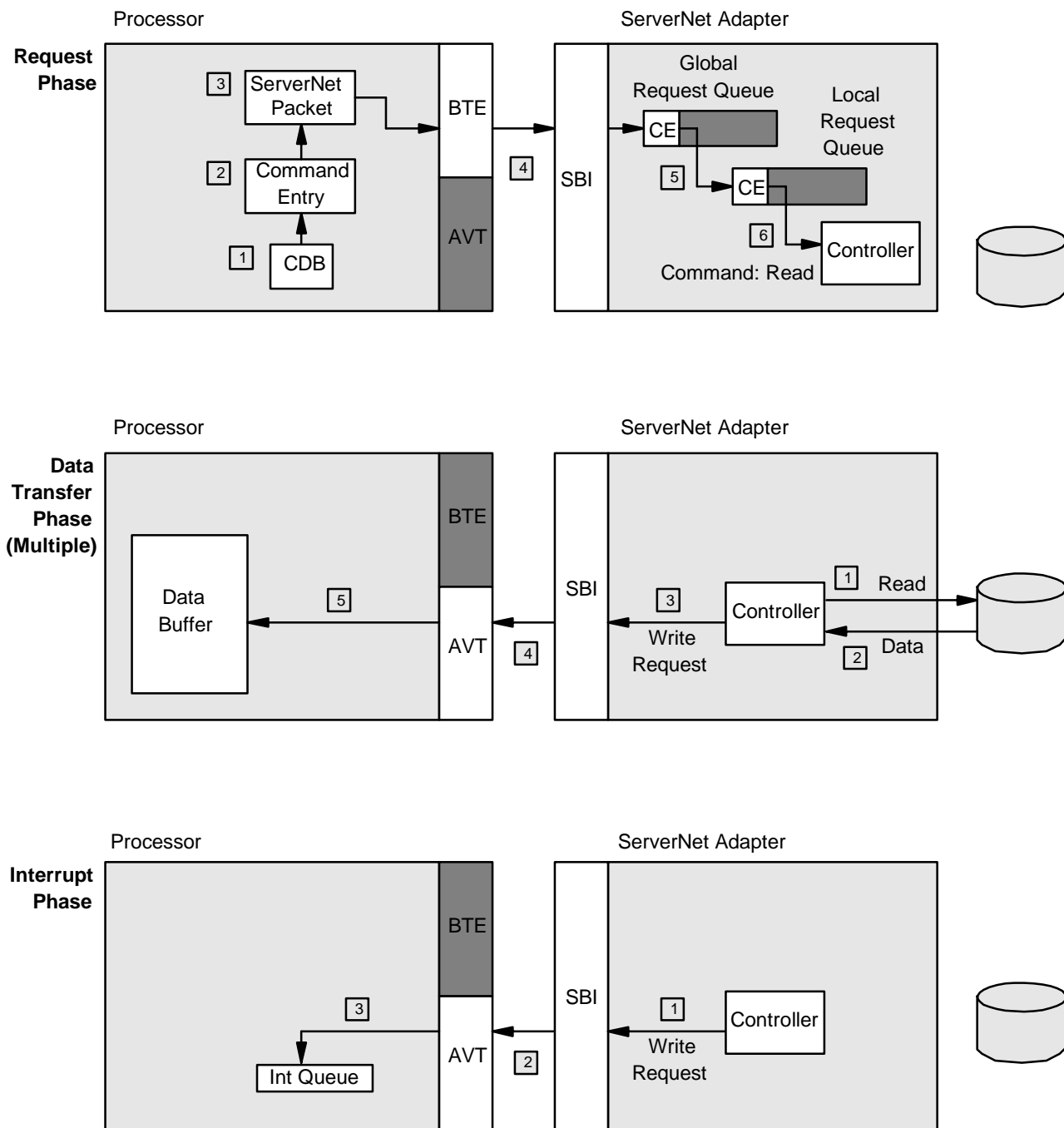
The module driver forwards the packet to ServerNet services, which transmits the packet to the ServerNet adapter (4), through the hardware of the block transfer engine (BTE) as a ServerNet write request. The ServerNet bus interface (SBI) returns a simple response packet back to the BTE and writes the packet data (the CE) into the addressed global request queue.

When the controller polls its global request queues, it finds the new request and transfers it to its local request queue (5). At this point, the controller discovers that the CDB it has received is requesting a read transfer (6).

In the **data transfer phase**, the controller commands the storage I/O device to read the requested data (1) and stores the received data in its own local memory (2). The controller then issues multiple write requests, along with the supplied ServerNet address of a client buffer, to the SBI (3). The SBI converts these controller requests to ServerNet packets and sends the packets through the ServerNet hardware (4) to the access validation and translation (AVT) logic in the processor.

The AVT logic validates the supplied ServerNet address, translates that address to a physical address, and stores the data of each received packet in the client's data buffer (5).

At the end of the data transfer, the controller initiates the **interrupt phase**. The controller creates an interrupt packet that it issues to the SBI as a ServerNet write request (1). For this packet, the controller uses a specific ServerNet address in processor memory that the module driver has allocated as an interrupt location. The AVT in the processor receives this packet (2) and puts it into an interrupt queue (3). When the interrupt is serviced, the I/O process is notified that the transfer has been completed.

**Figure 10-7. For Read Request, Controller Pushes Data in Write Transaction**

VST353.vsd

# Storage Write Request Processing

When an I/O process in a processor requests a write to a storage device, the only action that the processor initiates is to send the command to the controller. Thereafter, it is the responsibility of the controller to transfer the data, pulling the data to itself as if the controller is the originator of the request. The resultant sequence consists of three phases, illustrated in [Figure 10-8](#).

In the **request phase**, the module driver first of all assigns the data buffer a ServerNet address and makes an entry in the access validation and translation table (AVTT). Then the I/O process (1) builds a command descriptor block (CDB) and (2) embeds the CDB in a command entry (CE). The CE includes addresses of the client buffers of the data that the controller will pull later. Then the I/O process, through the XIO kernel, calls the module driver to create and send a ServerNet packet to a certain ServerNet addressable controller (3). The header of the created packet contains the destination and source ServerNet IDs, and the address field contains the ServerNet address of the next available slot in the appropriate global request buffer in the ServerNet adapter.

The module driver forwards the packet to ServerNet services, which transmits the packet to the ServerNet adapter (4), through the hardware of the block transfer engine (BTE) as a ServerNet write request. The ServerNet bus interface (SBI) returns a simple response packet back to the BTE and writes the packet data (the CE) into the addressed global request queue.

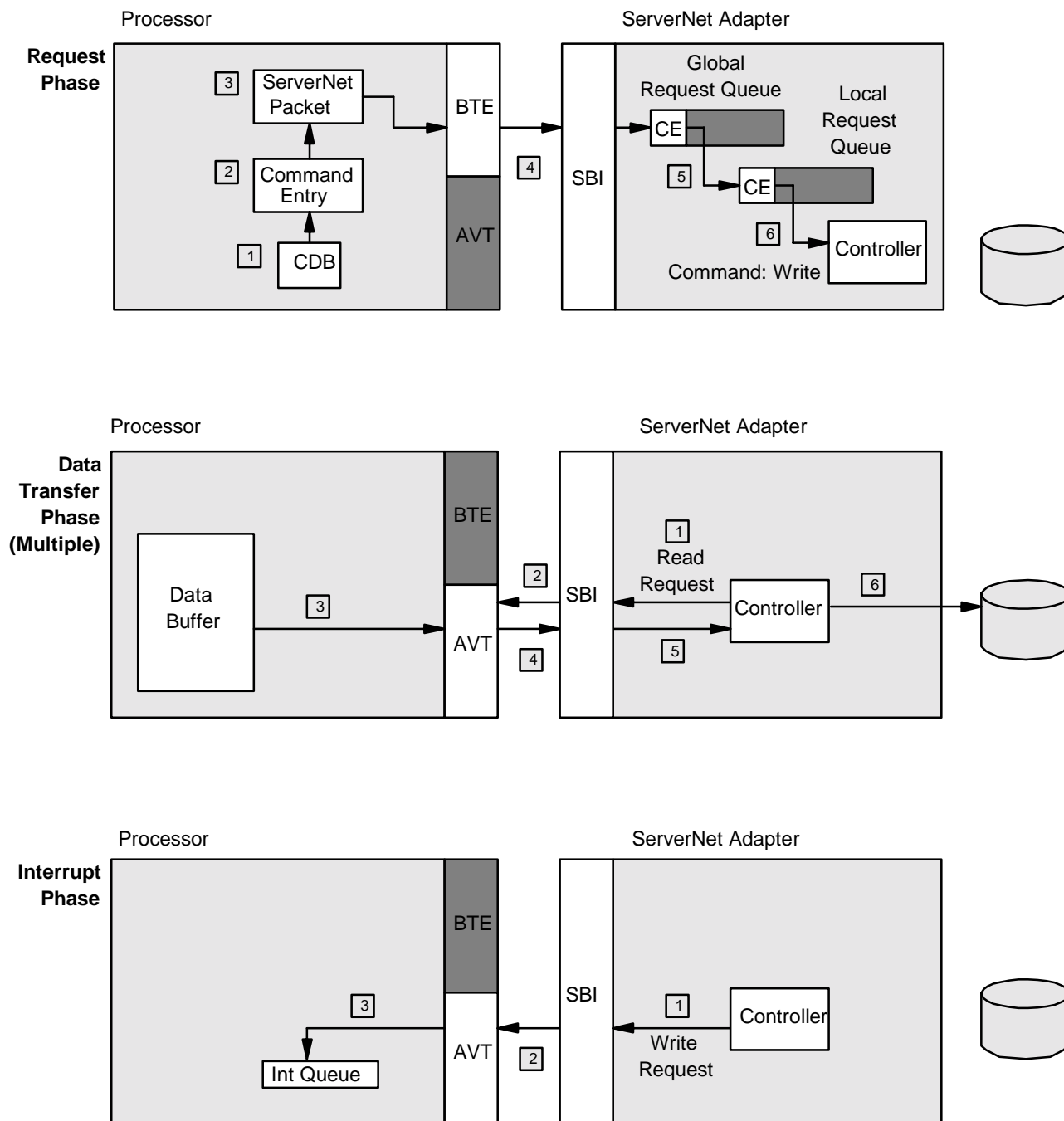
When the controller polls its global request queues, it finds the new request and transfers it to its local request queue (5). At this point, the controller discovers that the CDB it has received is requesting a write transfer (6).

In the **data transfer phase**, the controller initiates multiple read transactions (1) and forwards each read request packet to the ServerNet bus interface (SBI) for transmission through the ServerNet hardware to the access validation and translation (AVT) logic in the processor (2).

For each received packet, the AVT logic validates the supplied ServerNet address, translates that address to a physical address, and retrieves the requested data by DMA transfers, up to 64 bytes each time, from the client's data buffer (3). The AVT packages the data into ServerNet read response packets and transmits the packets back to the SBI (4).

The SBI supplies the packet data to the controller (5) to complete the read transactions. The controller thereafter commands the storage I/O device to write the data from the received packets (6).

At the end of the data transfer, the controller initiates the **interrupt phase**. The controller creates an interrupt packet that it issues to the SBI as a ServerNet write request (1). For this packet, the controller uses a specific ServerNet address in processor memory that the module driver has allocated as an interrupt location. The AVT in the processor receives this packet (2) and puts it into an interrupt queue (3). When the interrupt is serviced, the I/O process is notified that the transfer has been completed.

**Figure 10-8. For Write Request, Controller Pulls Data in Read Transaction**

VST354.vsd

# Communications Operation Queuing

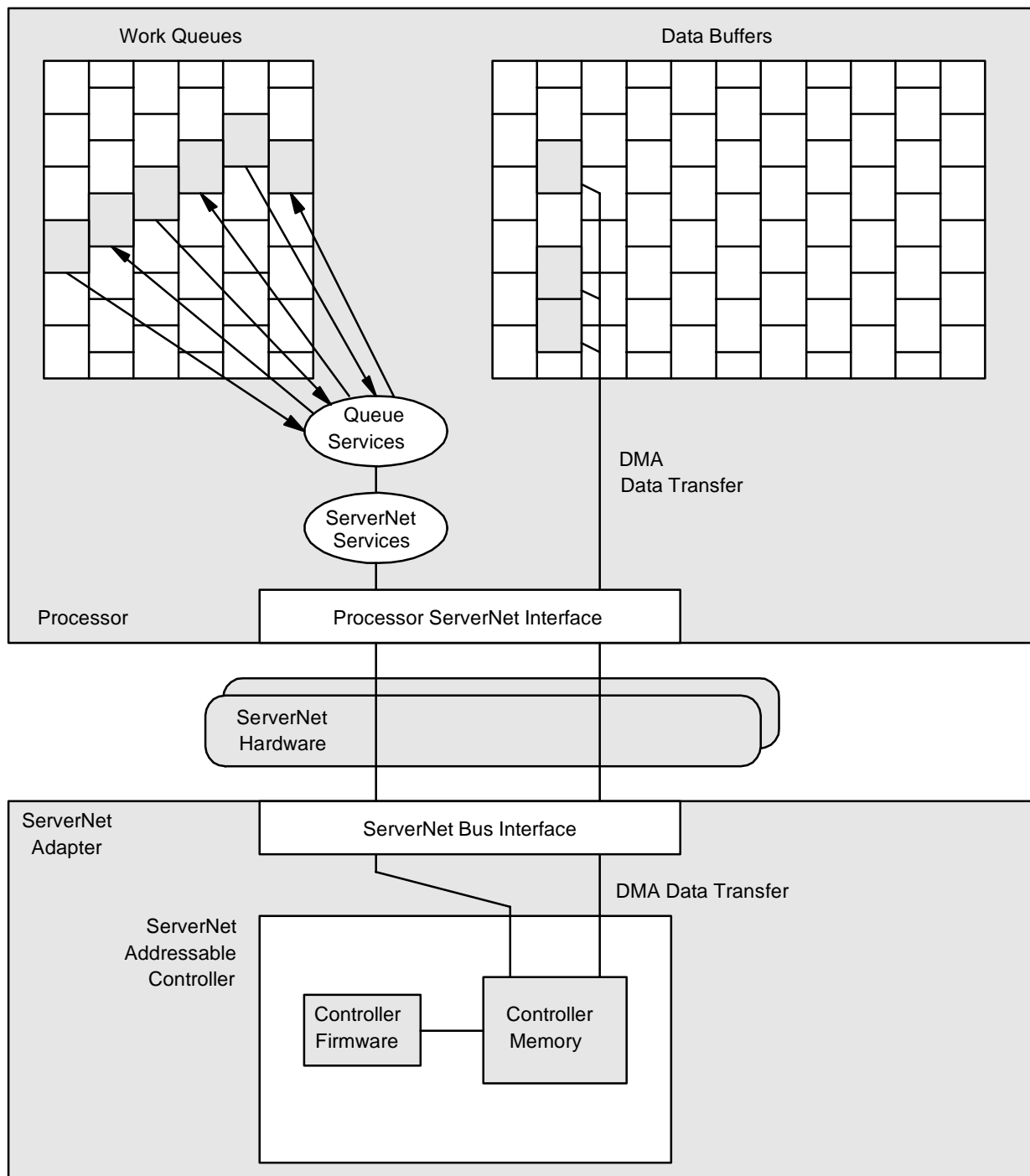
The remaining topics in this section describe communications I/O. The descriptions assume that you have read the first three topics in this section, which describe the basic differences between storage I/O and communications I/O. As illustrated earlier in [Figure 10-3](#) on page 10-7, communications I/O uses **work queues** (NIOC queues) that are located in processor memory. Most of the following information concerns the purpose and operation of these work queues, which are basically lists of I/O operations—such as request commands for reading or writing of data, or for delivery of addresses or status.

As shown in [Figure 10-9](#), processor memory has allocations both for work queues and for data buffers. A single resource, such as one communications line, can have several queues assigned and several data buffers of differing sizes. Queue information gets transferred between processor and controller memory, through the ServerNet hardware, under control of queue services in the processor and controller firmware in the controller. Data gets transferred between processor and controller memory by DMA engines in the processor ServerNet interface in the processor and the ServerNet bus interface (SBI) on a ServerNet adapter or multifunction I/O board (MFIOB).

Work queues are unidirectional. That is, a particular queue is used only for information that is being written by the controller and read by the processor, or that is being written by the processor and read by the controller. Unlike the queue entries used for storage I/O (which are of fixed size, 64 bytes each), entries in these communications work queues can be of varying length. The queues themselves may differ in length, one from another, but once created their length remains fixed. They are circular (first in, first out), and master copies of the head and tail pointers are maintained in the processor; the controller reads these master copies whenever it needs to know current settings, and can at times manipulate the head and tail pointers.



**Figure 10-9. Communications Controllers Use Work Queues That Are in Processor Memory**



VST355.vsd

# Application of Communications Queues

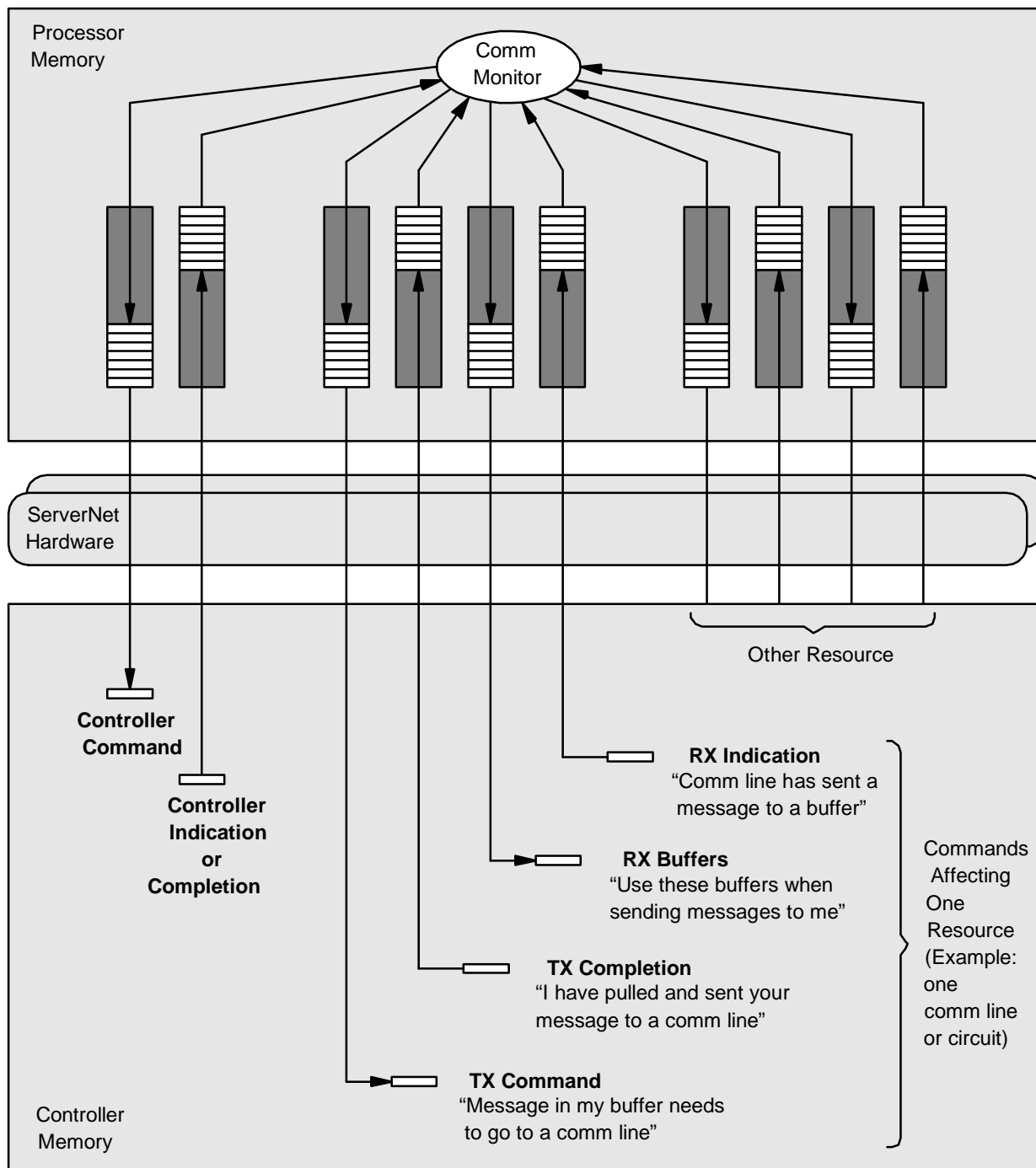
[Figure 10-10](#) illustrates the typical application of communications work queues. Note that the direction of the arrows indicates which entity (controller or communications monitor) writes to a given queue, and which entity reads from that queue. In the following discussion, the terms “out” and “in” refer to the perspective of the processor—command information going **out** from the processor or coming **in** to the processor.

Three groupings of queues are shown. The first group is a single pair (out and in). The out-direction queue provides a means for a communications monitor to transfer command information to a particular controller (for example to provide addressing information to the controller regarding all the other queues available to that controller). The in-direction queue provides a means for the controller to signify to the monitor such things as completion of an earlier request, or to supply new information (such as ServerNet addresses allocated in the controller).

The next two groups (one of which is shown detailed) are representative of what could be numerous other groupings of queues. Typically, these queues are allocated on the basis of two pairs for each resource (a resource being typically one communications line or one ATM virtual circuit). One pair of the group (out and in) is used for transmission (TX) operations, usually commands in the out direction and completion information in the in direction. The other pair of the group (out and in) is used for receiving (RX) operations, usually to provide buffer addresses to the controller in the out direction and indication of new controller actions in the in direction.

The figure paraphrases typical exchanges that occur between the monitor and the controller. A TX command typically informs the controller that a message exists in a data buffer at a certain ServerNet address in the processor and that it needs to go to a certain communications line. After the controller has read the command and acted upon it, it sends a TX completion message to the monitor that essentially says that the controller has pulled the message from the data buffer and successfully transferred it to the specified communications line.

An RX buffer command from the monitor typically gives the controller a selection of data buffer addresses to use whenever it needs to send a message to the monitor. When the controller actually does need to send a message to the monitor, it first goes ahead and sends the message with ServerNet DMA transfers and then, after the message has been sent, informs the monitor which buffer it used by supplying the ServerNet address in an RX indication command.

**Figure 10-10. Communications Queues Are Unidirectional and Paired**

VST356.vsd

# Typical Use of Queue Pairs

[Figure 10-11](#) illustrates the typical way in which pairs of queues work. The illustration is applicable to both transmission from the processor (TX) and reception by the processor (RX). However, the application differs in the TX and RX cases, so they are described separately in the following paragraphs. Be aware that all manipulation of the queues is accomplished by calls to queue services (see [Figure 10-3](#) on page 10-7 for orientation). Also, it is queue services that updates the queue pointers as entries are written in and are later released after being read and acted upon.

Note that, for the outbound queues (TX command queue or RX buffer queue), the module driver manipulates the write pointer, and the controller manipulates the read pointer. For the inbound queues (TX completion queue or RX indication queue), the controller manipulates the write pointer, and the module driver manipulates the read pointer. For the controller to write entries into queues, the controller uses a local copy of the pointers and periodically pulls fresh copies to itself across ServerNet hardware.

In the case of a TX operation, the queues involved are the TX command queue for outbound messages and the TX completion queue for inbound messages.

To begin the TX operation, the module driver, through queue services, writes a command entry into TX command queue for some particular communications resource (1). This new entry is designated in the figure with an *x*. Queue services updates the write pointer to the next available entry. As the controller reads and acts upon prior entries in its queue for this resource, it eventually comes upon the entry marked *x* (2). (The write pointer has been advancing as other entries have been added and has wrapped around the physical end of the queue.) The controller pulls the command entry to itself, updates the TX command read pointer, and then typically enters into a DMA transfer of the message data.

After the data transfer is completed, the controller writes an entry into the TX completion queue (marked *xx*), describing the status of the data transfer (3). The controller moves the write pointer of the TX completion queue. As the module driver reads and acts upon prior entries in the TX completion queue, it eventually comes to the *xx* entry and reads the status information it contains (4), and the I/O operation is completed. Queue services advances the read pointer to eliminate the *xx* entry.

In the case of an RX operation, the queues involved are the RX buffers queue for outbound messages and the RX indication queue for inbound messages. In this case, the module driver has set up a buffer pool and begins the operation by writing an entry into the RX buffers queue (1). This entry, marked *x*, describes the ServerNet addresses and the lengths of the buffers in the pool. When the controller comes to the *x* entry (2), it pulls the contents to itself and saves the information about the available buffers.

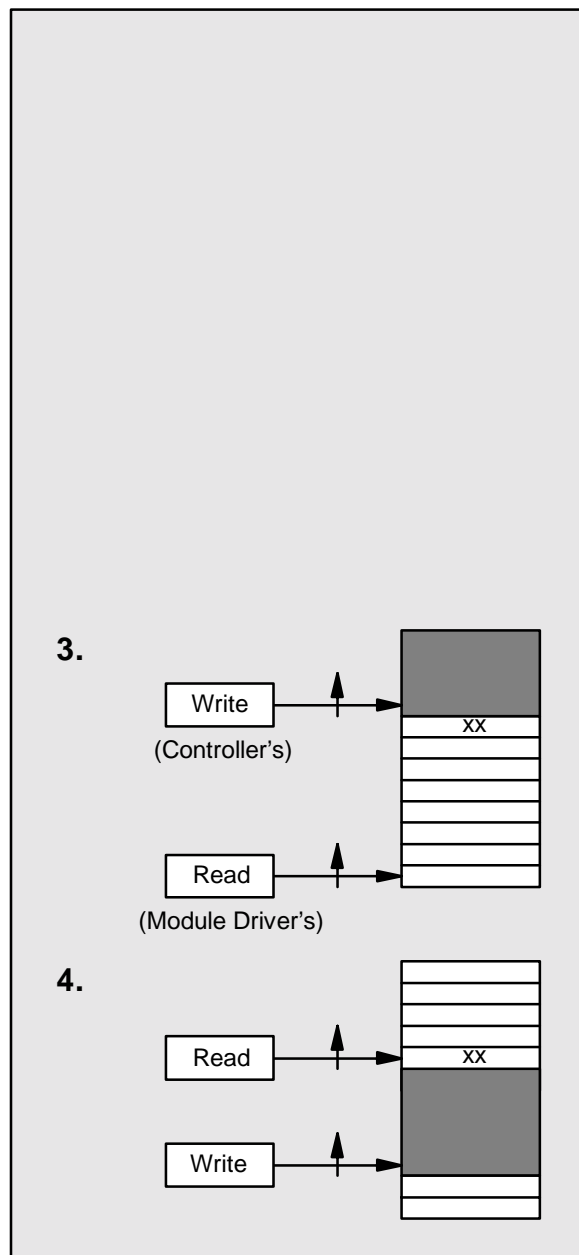
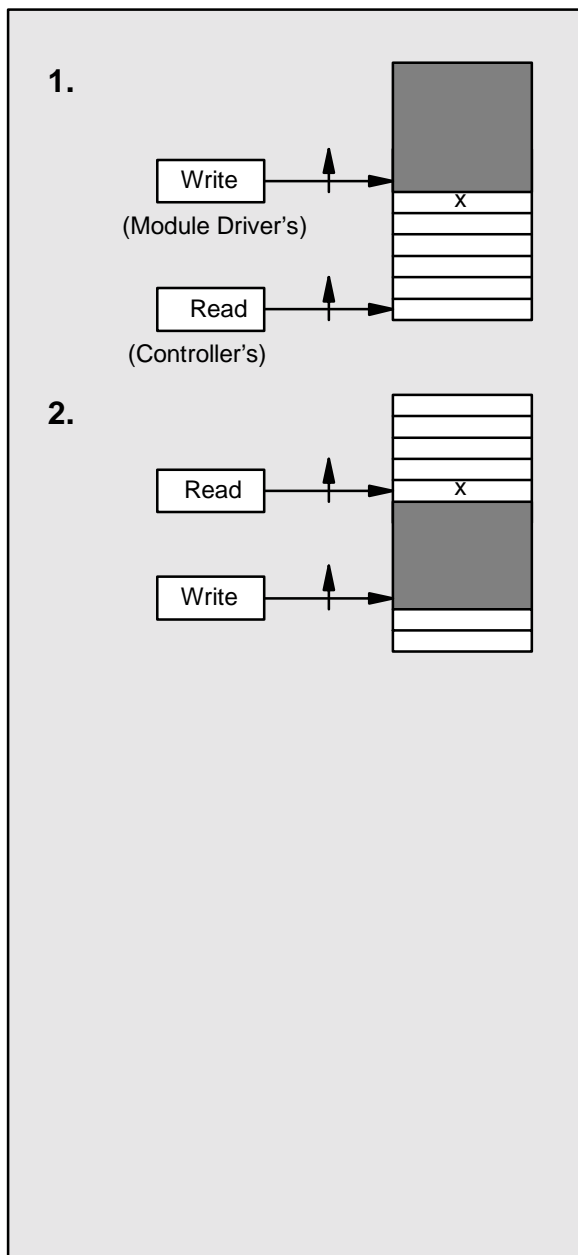
At some later time, the controller receives some inbound message data. Without any other preparation, the controller selects a buffer from the buffer pool and enters into a DMA transfer of the message data. When the transfer is completed, the controller writes an entry into the RX indication queue (3). This entry (*xx*) lets the monitor know that data has been transferred into some particular buffer, specified by a ServerNet

address. When the module driver reads this entry (4), it notifies the appropriate client, and the operation is completed. Queue services advances the read pointer to eliminate the xx entry.

**Figure 10-11. Monitor Writes in Outbound Queues, Module Driver Writes in Inbound Queues**

**Outbound:** TX Command Queue  
or  
RX Buffers Queue

**Inbound:** TX Completion Queue  
or  
RX Indication Queue



VST357.vsd

## Actions for Empty or Full Queues

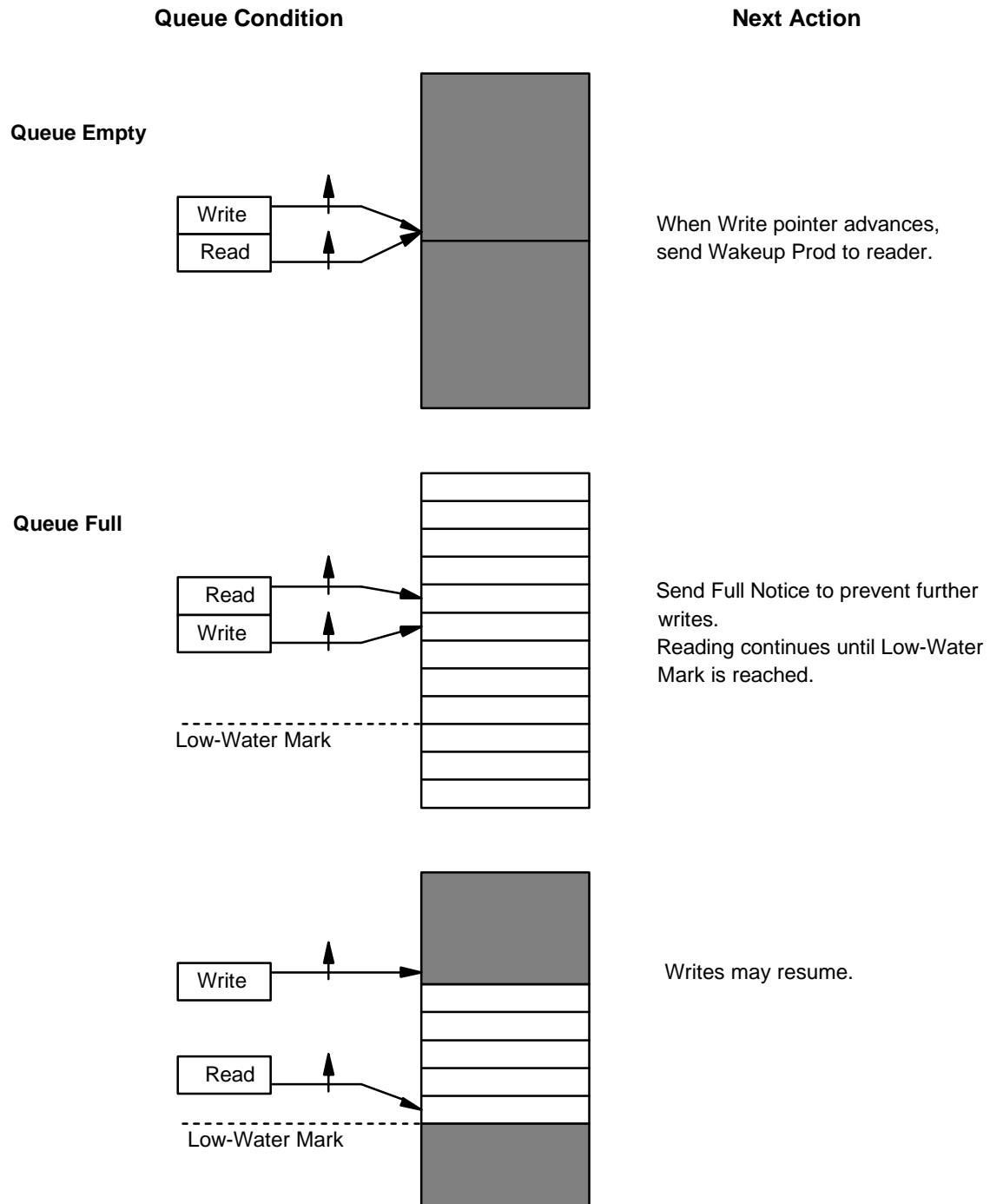
In all of the actions described in the preceding topic, the read pointer is essentially chasing the write pointer. That is, the writer of entries into a queue is adding new entries to the head of the queue at some unpredictable, irregular rate, and the reader of those entries is acting upon the entries at the tail of the queue, trying to catch up to the most recently added entry. Ideally, these operations remain in balance and are self-adjusting to a large degree. However, there are times when the read pointer does catch up to the write pointer (the queue is empty), and there are times when the read pointer falls so far behind that the write pointer, having wrapped around, catches up to the read pointer (the queue is full). These two situations require processor or controller interrupts for special handling. These situations are illustrated in [Figure 10-12](#) and are described here.

For the **queue-empty condition**, illustrated in the top part of [Figure 10-12](#), both the read pointer and the writer pointer are pointing to the same location in the queue. Because it would be impractical for the controller to keep checking across the ServerNet hardware for a change in this situation, the protocol is for queue services to send a **wakeup prod** to the queue reader (the controller in some cases, a client process in other cases). This prod consists of a single ServerNet packet sent to a predetermined ServerNet address in an interrupt queue in the reader's memory. When that interrupt packet is read, the controller or client is notified that there is a new entry in the queue. The reader then gets the entry at the current setting of the read pointer, and normal operation resumes.

For the **queue-full condition**, illustrated in the lower two thirds of [Figure 10-12](#), the write pointer has wrapped around and caught up to the read pointer. This situation causes the reader of that queue to send a full notice to the reader. The full notice is a single ServerNet packet that informs the reader that the queue cannot accept any more entries.

Although it is possible for the reader to resume writing as soon as one location is freed (the read pointer advances by one), a low-water facility is available. The **low-water mark** is a configurable parameter that allows the queue to acquire several empty locations before the writer to resumes writing to the queue. The availability of these empty locations can restore the read-write balance and minimize the necessity to send frequent full notices across the ServerNet hardware.

**Figure 10-12. Empty Queue Requires Wakeup Prod, Full Queue Requires Full Notice**



VST358.vsd

# Communications Request Processing

As stated back in the first topic of this section, transfers that are inbound to the host server originate externally by some request from a workstation that is part of an external network. Transfers that are outbound from the host server typically provide information or control from some service in the NonStop processor. In either case, the request is communicated to a ServerNet addressable controller, which assumes control of the transfer. The operations that then follow consist of pushing and pulling queue entries to and from a work queue, and pushing and pulling data to and from data buffers. [Figure 10-13](#) illustrates these four operations.

When the controller needs to **pull a queue entry** (or the queue pointers), it issues a read request (1) to the ServerNet bus interface (SBI) that is located on its particular ServerNet adapter (or MFIOB). It knows the ServerNet address of the entry by its own local copy of the queue pointers or other saved information (the location of the pointers, for example). The SBI then sends a ServerNet read request packet through the ServerNet hardware (2) to the access validation and translation (AVT) logic in the processor. The AVT logic validates the supplied ServerNet address, translates that address to a physical address, and retrieves the entry from the work queue (3). The AVT packages the entry into a ServerNet response packet and sends it to the SBI (4) as part of the same ServerNet transaction. The SBI, expecting this response, forwards the packet data (the queue entry) to the controller (5), as requested.

When the controller needs to **push a queue entry** (or queue pointer), it issues a write request (1) to the ServerNet bus interface (SBI) that is located on its particular ServerNet adapter (or MFIOB). The SBI then sends a ServerNet write request packet through the ServerNet hardware (2) to the access validation and translation (AVT) logic in the processor. The AVT logic sends a simple write response back to the SBI, validates the supplied ServerNet address, translates that address to a physical address, and writes the entry into the work queue (3).

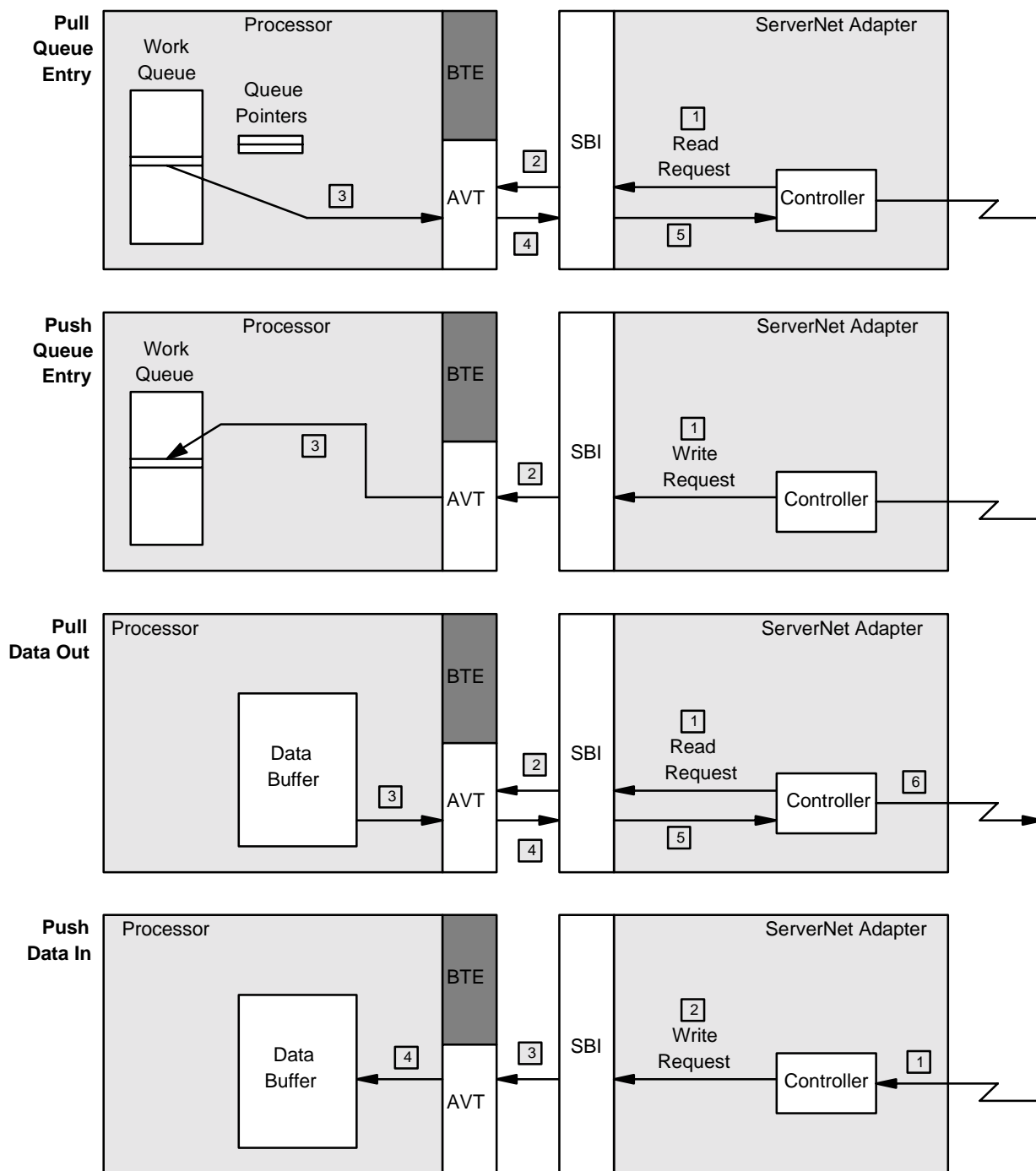
When the controller needs to **pull data** from a data buffer in the processor, it issues a read request (1) to the ServerNet bus interface (SBI). The SBI then sends a ServerNet read request packet through the ServerNet hardware (2) to the access validation and translation (AVT) logic in the processor. The AVT logic validates the supplied ServerNet address, translates that address to a physical address, and retrieves the data from the data buffer (3). The AVT packages the data into a ServerNet response packet and sends it to the SBI (4) as part of the same ServerNet transaction. The SBI, expecting this response, forwards the packet data to the controller (5), as requested. The controller, when it has accumulated the full message from individual ServerNet packets, then transmits the data to the external source that requested the data (6).

When the controller needs to **push data** that has come in from some external source (1), it issues a write request (2) to the SBI. The SBI then sends a ServerNet write request packet through the ServerNet hardware (3) to the access validation and translation (AVT) logic in the processor. The AVT logic sends a simple write response back to the SBI, validates the supplied ServerNet address, translates that address to a physical address, and writes the packet data into the data buffer (4).



Note that the bus transfer engine (BTE) is not involved in any of these operations, because none of the work originates in the processor.

**Figure 10-13. Controller Originates All Work in Most Communications Transfers**



VST359.vsd



# 11 TNS Instruction Set

The TNS instruction set of the NonStop S-series processor consists of approximately 280 machine instructions. This section provides text descriptions of these instructions, with the exception of a few that are reserved for operating system use. Diagrams are also included showing the action of some of the more commonly used instructions.

In addition, [Appendix A, TNS Instruction Lists](#) and [Appendix B, TNS Instruction Binary Coding](#) provide a number of useful reference tables pertaining to the instruction set. [Appendix C, TNS Instruction Set Definition](#) defines the instructions in a pseudocode form.

Definition of the RISC instruction set is not included in this manual.

The general information in the first four topics applies in common to many of the instructions defined in the remainder of the section. The information should be helpful in visualizing details of the ways in which the various instructions actually work. Where appropriate, the instruction definitions refer back to the illustrations given here.

Additional figures later in this section are positioned near to the instruction definitions that they illustrate. Usually such figures apply to only one instruction or a few that are grouped together.

The instruction definitions are given under the following major headings:

[Memory Addressing Instructions](#)

[Immediate Operand and Shift Instructions](#)

[Boolean Instructions Operate on Stack Registers](#)

[Move, Compare, and Scan Instructions](#)

[Definitions of TNS Instructions](#)

[Additional Operating-System-Only Instructions](#)

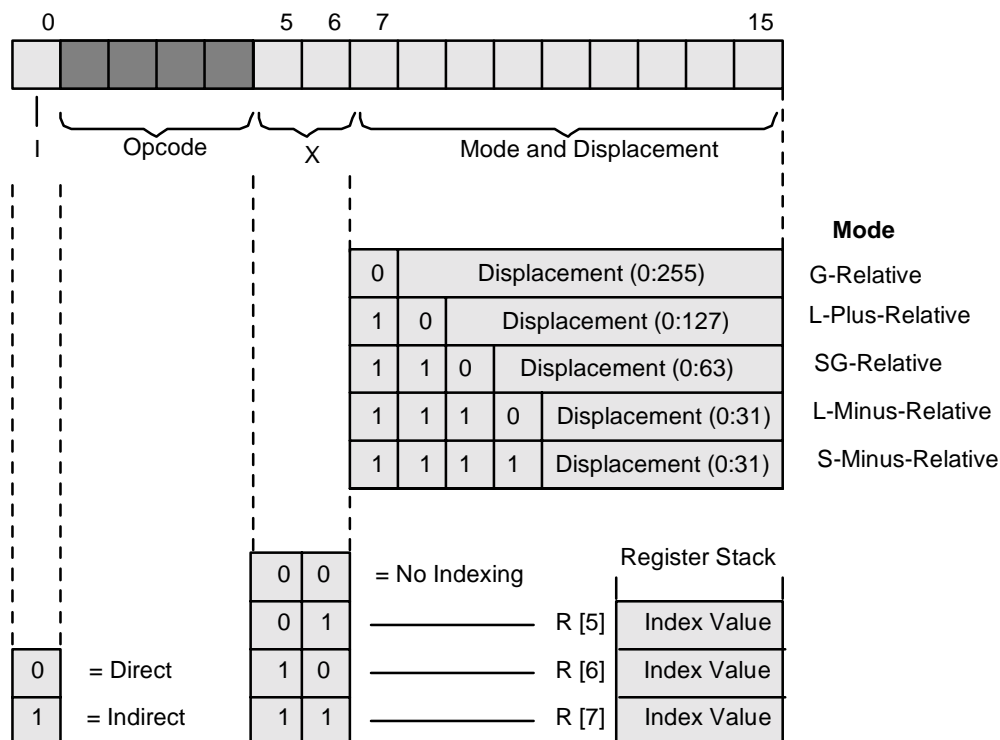
# Memory Addressing Instructions

Figure 11-1 shows the instruction word format for memory data reference instructions that refer to single-length (16-bit) operands. There are ten such instructions: LDX, NSTO, LOAD, STOR, LDB, STB, LDD, STD, LADR, and ADM.

For both the single-length and doubleword formats, bit 7 begins the mode and displacement field. The 9-bit field of I.<7:15> provides different displacement ranges depending on the mode selected. Four of the modes are with reference to the data segment, and the other (SG-relative) is with reference to the system data segment.

I.<5:6> is the indexing field, specifying either no indexing or a code that identifies one of three stack registers that contains the index value. I.<0> is the direct/indirect specifier.

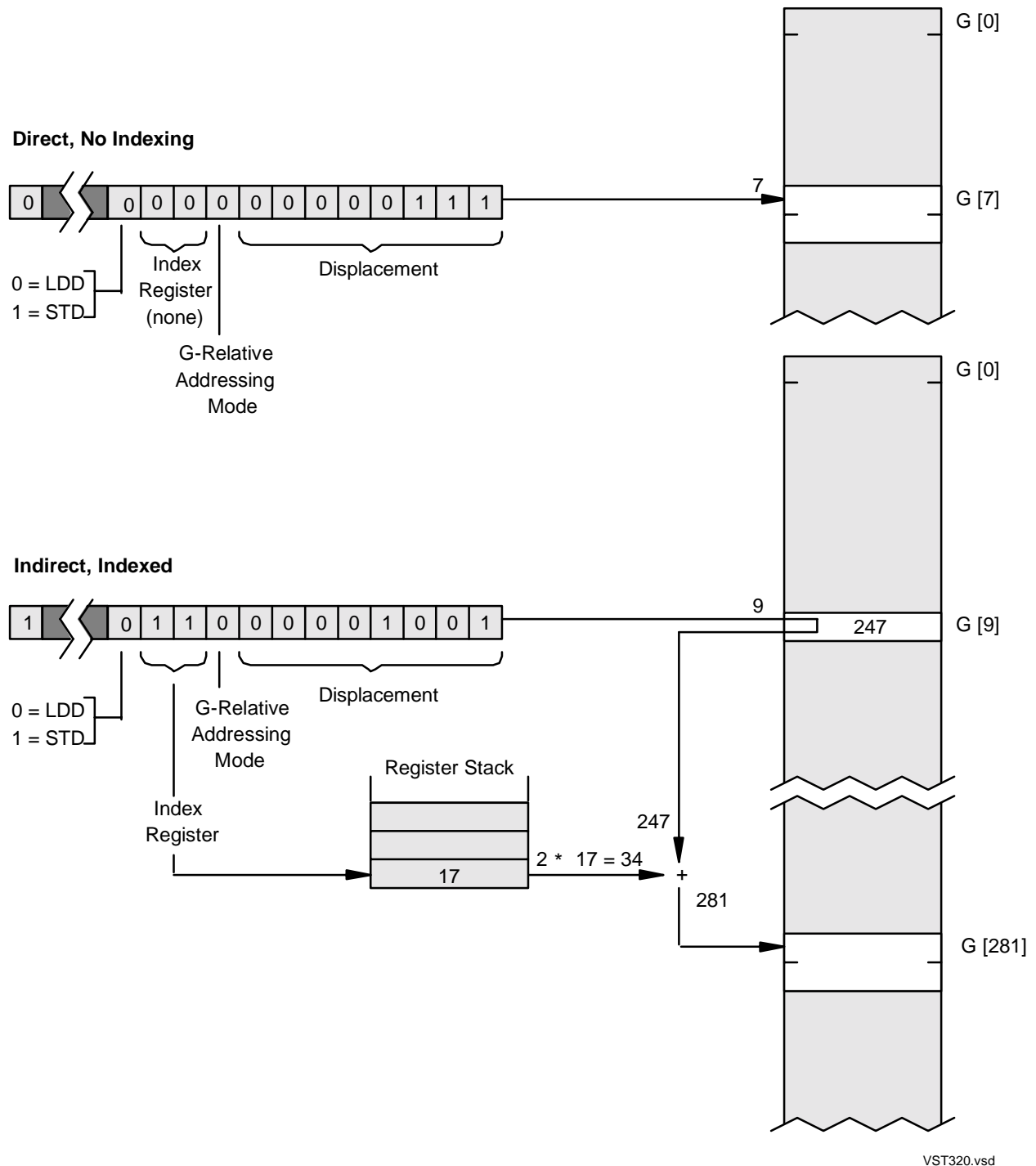
**Figure 11-1. Memory Addressing Instructions Provide Access to the Data Segment**



VST319.vsd

[Figure 11-2](#) illustrates the addressing methods for the doubleword instructions LDD and STD. The upper example illustrates LDD (Load Doubleword) with direct addressing and no indexing. The lower example illustrates LDD with indirect addressing and indexing.

**Figure 11-2. Examples of Doubleword Addressing**

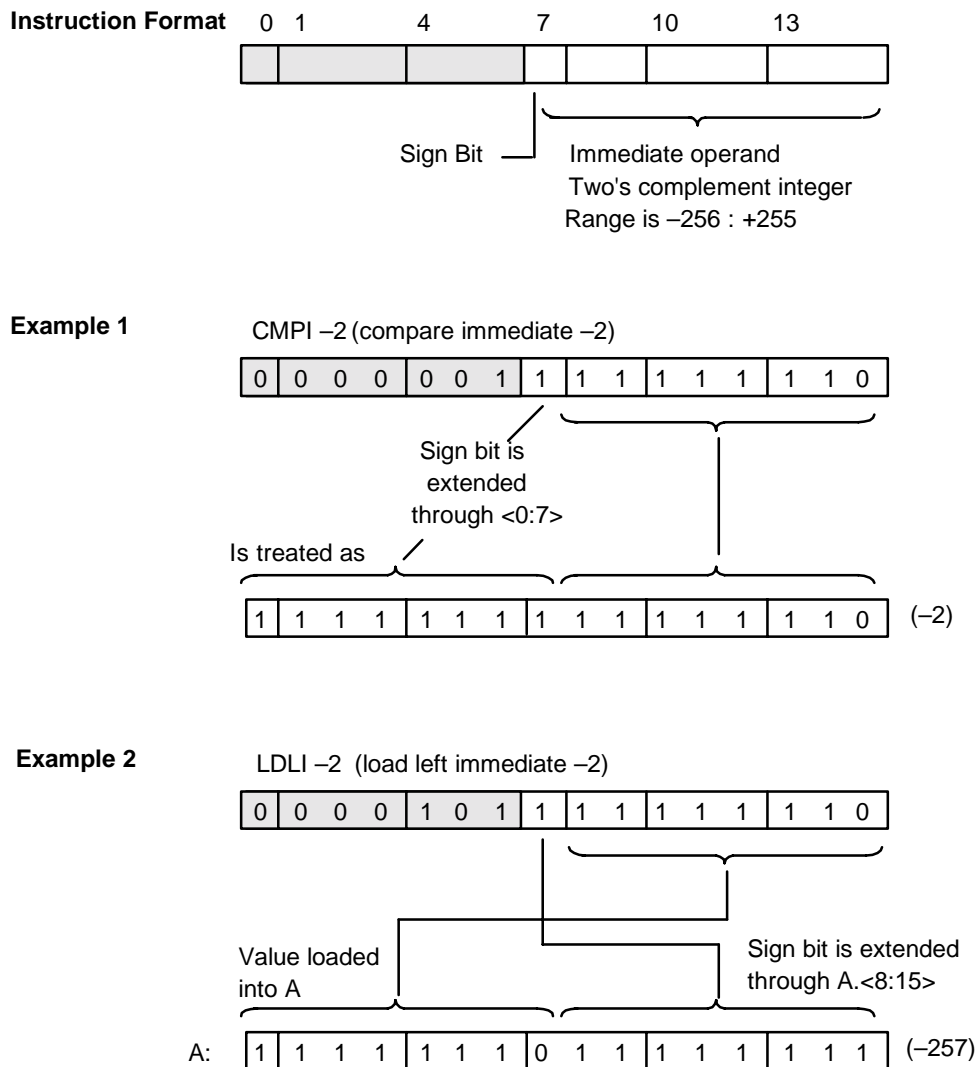


# Immediate Operand and Shift Instructions

[Figure 11-3](#) illustrates the general instruction word format for instructions that use immediate operands. Immediate operand instructions operate on stack registers.

The first example shows how an 8-bit immediate value is extended to a 16-bit value for comparison purposes. The second example shows the result of a load-left operation; the sign is extended into the right eight bits.

**Figure 11-3. Examples of Immediate Operand Instructions**



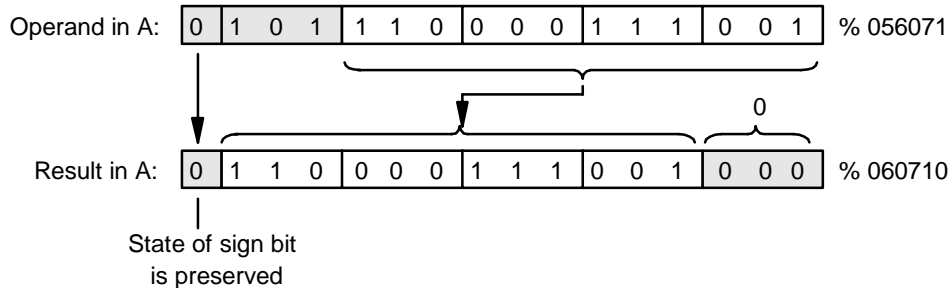
VST321.vsd

[Figure 11-4](#) presents a comparison of logical (unsigned) shifts and arithmetic (signed) shifts. In arithmetic left shifts, the sign bit is unaffected (except in accelerated mode); in logical left shifts, the sign bit participates in the shift. In arithmetic right shifts, the sign bit is propagated to the right; in logical right shifts, vacated positions are filled with zeros.

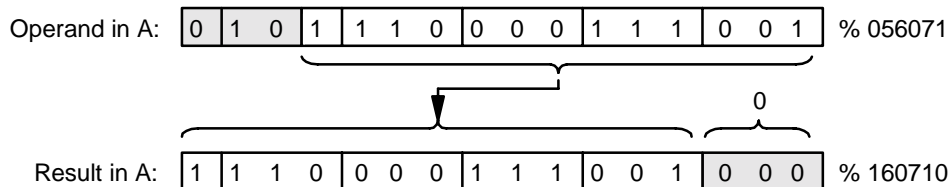
**Figure 11-4. Examples of Logical and Arithmetic Shifts**

#### Left Shifts

ALS 3 (Arithmetic left shift three positions)

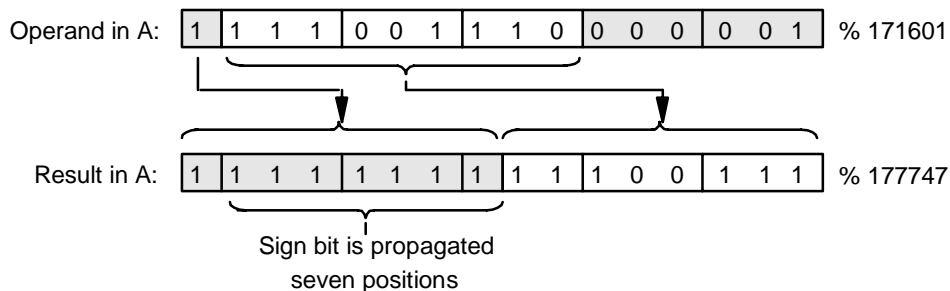


LLS 3 (Logical left shift three positions)

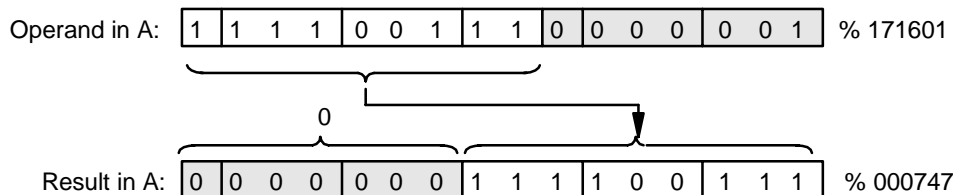


#### Right Shifts

ARS 7 (Arithmetic right shift seven positions)



LRS 7 (Logical right shift seven positions)



VST322.vsd

# Boolean Instructions Operate on Stack Registers

[Figure 11-5](#) shows examples of the four basic Boolean instructions: LAND, LOR, XOR, and NOT. In these examples of Boolean operations, both operands are on the top of the register stack. The instructions delete the operands and leave the result on the register stack, in A.

**Figure 11-5. Examples of Boolean Instruction Operations**

LAND (Logical AND)	$0 + 0 = 0$	<table><tr><td></td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>		0	1	1	0	1	1	Operand 1
		0	1	1	0	1	1			
	$0 + 1 = 0$									
	$1 + 0 = 0$	<table><tr><td></td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>		0	0	1	1	1	0	Operand 2
	0	0	1	1	1	0				
$1 + 1 = 1$	<table><tr><td></td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>		0	0	1	0	1	0	Result	
	0	0	1	0	1	0				
LOR (Logical OR)	$0 + 0 = 0$	<table><tr><td></td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>		0	1	1	0	1	1	Operand 1
		0	1	1	0	1	1			
	$0 + 1 = 1$									
	$1 + 0 = 1$	<table><tr><td></td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>		0	0	1	1	1	0	Operand 2
	0	0	1	1	1	0				
$1 + 1 = 1$	<table><tr><td></td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>		0	1	1	1	1	1	Result	
	0	1	1	1	1	1				
XOR (Exclusive OR)	$0 + 0 = 0$	<table><tr><td></td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>		0	1	1	0	1	1	Operand 1
		0	1	1	0	1	1			
	$0 + 1 = 1$									
	$1 + 0 = 1$	<table><tr><td></td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>		0	0	1	1	1	0	Operand 2
	0	0	1	1	1	0				
$1 + 1 = 0$	<table><tr><td></td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>		0	1	0	1	0	1	Result	
	0	1	0	1	0	1				
NOT (One's complement)	$0 \rightarrow 1$	<table><tr><td></td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>		0	1	1	0	1	1	Operand
		0	1	1	0	1	1			
$1 \rightarrow 0$	<table><tr><td></td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>		1	0	0	1	0	0	Result	
	1	0	0	1	0	0				

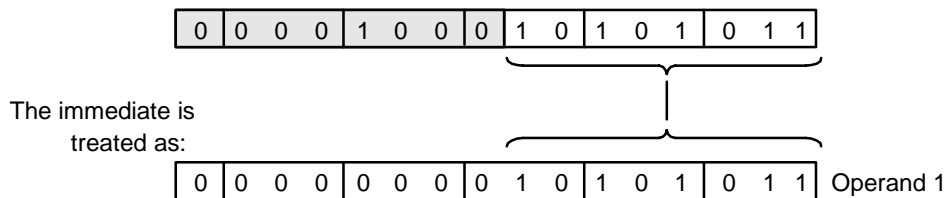
VST323.vsd



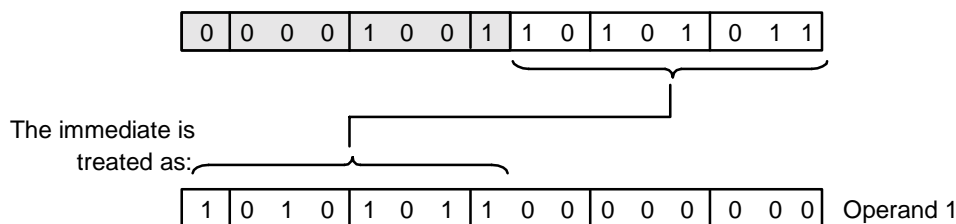
Similarly, [Figure 11-6](#) shows examples of the four Boolean instructions that use immediate operands: ORRI, ORLI, ANRI, and ANLI. These examples show how the 8-bit or 9-bit immediate operand is expanded to a 16-bit operand for the four immediate Boolean instructions. Only the immediate operand is shown here (operand 1); the second operand is in A on the register stack.

**Figure 11-6. Examples of Boolean Immediate Instruction Operations**

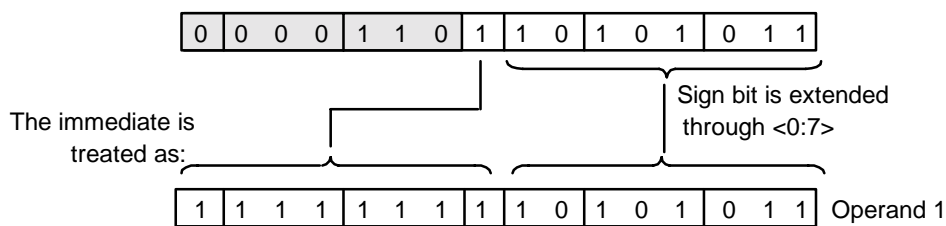
ORRI (OR right immediate)



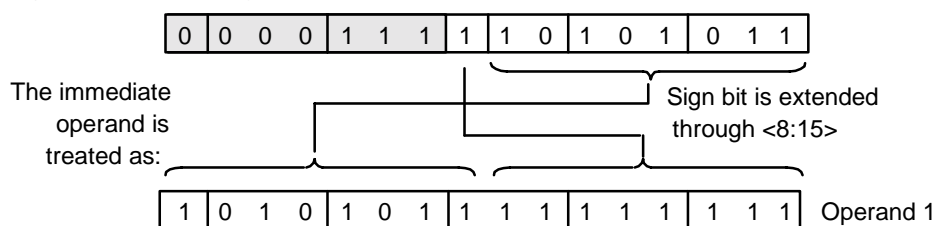
ORLI (OR left immediate)



ANRI (AND right immediate)



ANLI (AND left immediate)



VST324.vsd

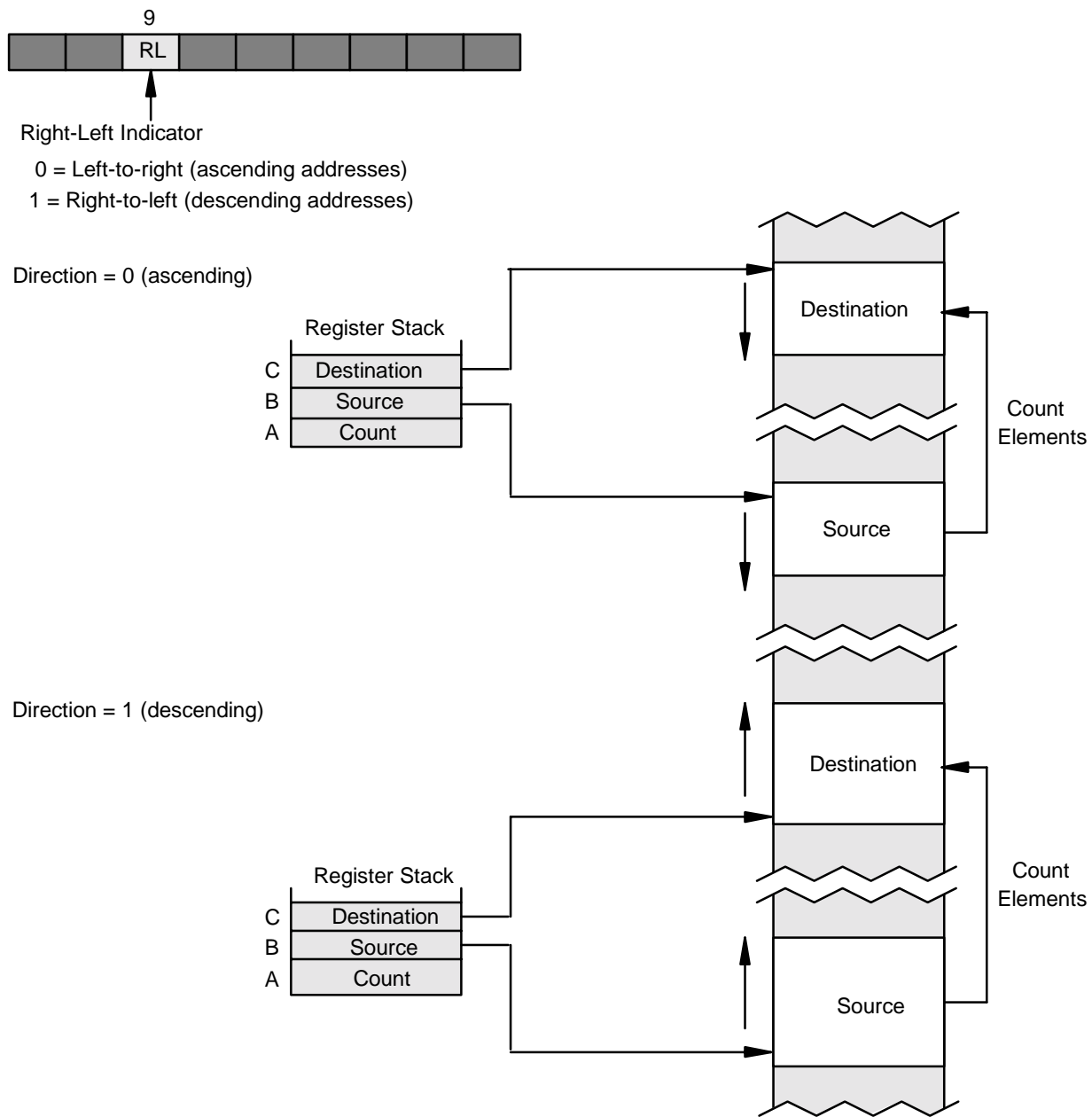
# Move, Compare, and Scan Instructions

One bit in the instruction word format of nonextended memory-to-memory move, compare, and scan instructions specifies an ascending order or descending order of addresses used for the source and destination of the moved data. Starting addresses for the destination and the source are specified in the register stack, as well as the count of elements to be moved.

[Figure 11-7](#) provides a comparison of ascending and descending moves, compares, and scans, as described in the instruction definitions for MOVW, MOVB, COMW, COMB, SBW, and SBU.

Note that bit 9 of the instruction word format specifies ascending order of addresses if 0 and descending order if 1. Elements are 16-bit words for MOVW and COMW (Move Words and Compare Words). Elements are bytes for MOVB, COMB, SBW, and SBU (Move Bytes, Compare Bytes, Scan Bytes While, and Scan Bytes Until).

The upper example shows an ascending move (or compare or scan). Register A specifies the count of elements involved, B specifies the source address, and C specifies the destination address. These values are updated as the operation progresses.

**Figure 11-7. Moves Can Be Ascending or Descending**

VST325.vsd

# Definitions of TNS Instructions

The instruction definitions given on this and the following pages are listed in alphabetical order to facilitate quick reference. A listing by general category is provided in [Table A-2](#) on page A-9. Unless otherwise stated, “stack” refers to the register stack.

**ADAR (00016-).** Add A to a Register. A is added in signed integer form to the register pointed to by the Register field of the instruction. A is deleted from the stack. Overflow is set if the result is greater than 32767 or less than –32768. Carry can be set, and Condition Code is set on the result. For binary coding details, refer to [Table B-5](#) on page B-5.

**ADDI (104---).** Add Immediate Operand to A. The immediate operand is added to A in signed integer form. Overflow is set if the result is greater than 32767 or less than –32768. Carry can be set. Condition Code is set. Examples of the use of immediate operands are shown in [Figure 11-3](#) on page 11-4.

**ADDs (002---).** Add Immediate Operand to S. The signed immediate operand is added to the S register in integer form. If the resultant value in S is greater than 32767, a stack overflow trap occurs.

**ADM (-74---).** Add A to Variable in Data Space. The A register is added in integer form to the contents of the effective memory location and the Condition Code is set on the sum. Overflow is set if the result is greater than 32767 or less than –32768. Carry can also be set. A is then deleted from the stack. For binary coding details, refer to [Table B-1](#) on page B-1.

**ADRA (00014-).** Add Register to A. The contents of the register pointed to by the Register field of the instruction are added in integer form to register A. Overflow is set if the result is greater than 32767 or less than –32768. Carry can be set, and Condition Code is set on the result. For binary coding details, refer to [Table B-5](#) on page B-5.

**ADXl (104---).** Add Immediate Operand to an Index Register. The immediate operand is added in signed integer form to the contents of the index register specified by the “x” field of the instruction. Overflow is set if the result is greater than 32767 or less than –32768. Carry can be set; Condition Code is set on the result. For binary coding details, refer to [Table B-2](#) on page B-2.

**ALS (0302--).** Arithmetic Left Shift. If the shift count field is zero, the word contained in B is shifted left by the dynamic count contained in A. A is then deleted from the stack. However, if shift count field is not zero, A is shifted left by that number. The sign bit is preserved only in nonaccelerated mode; in accelerated mode, ALS is treated as LLS (Logical Left Shift). On single-word shifts, dynamic shift counts greater than 31 or less than 0 give undefined results. Condition Code is set. Overflow and Carry are unaffected. Refer to [Figure 11-4](#) on page 11-5 for a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

ANG (000044). AND to Memory. The word in B is logically ANDed to a word in the current data segment that is specified by a 16-bit address in A. The result remains in the data segment location, and A and B are deleted from the stack. Condition Code is set.

ANLI (007---). AND Left Immediate Operand With A. The 8-bit immediate operand is shifted left eight places, the sign bit is propagated into the low-order bits, and the resulting integer is logically ANDed to A. Condition Code is set. (For binary coding details, refer to [Table B-2](#) on page B-2; for an example, see [Figure 11-6](#) on page 11-7.)

ANRI (006---). AND Right Immediate Operand to A. The 8-bit immediate operand is extended to 16 bits by propagating the sign into the high-order bits, and the resulting integer is logically ANDed to A. Condition Code is set. (For binary coding details, refer to [Table B-2](#) on page B-2; for an example, see [Figure 11-6](#) on page 11-7.)

ANS (000034). AND to SG Memory. The word in B is logically ANDed to a word in the system data segment that is specified by a 16-bit address in A. The result remains in the system data location, and A and B are deleted from the stack. Condition Code is set. This is a privileged instruction.

ANX (000046). AND to Extended Memory. The word in C is logically ANDed to a word in memory that is specified by a 32-bit even-byte address in BA. The result remains in the memory location, and A, B, and C are deleted from the stack. Condition Code is set.

ARS (0303--). Arithmetic Right Shift. If the shift count field is zero, the word contained in B is shifted right, propagating the sign bit, by the dynamic count contained in A. A is then deleted from the stack. However, if the shift count field is not zero, A is shifted right, propagating the sign bit, by that number. On single-word shifts, dynamic shift counts greater than 31 or less than 0 give undefined results. Condition Code is set. Refer to [Figure 11-4](#) on page 11-5 for a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

BANZ (-154--). Branch on A Not Zero. If the A register is nonzero, a direct or indirect branch is taken (depending on the “i” field of the instruction). If the A register equals zero, the next instruction is executed. In either case, A is deleted from the stack. For binary coding details, refer to [Table B-4](#) on page B-4.

BAZ (-144--). Branch on A Zero. If the A register equals zero, a direct or indirect branch is taken (depending on the “i” field of the instruction). If the A register does not equal zero, the next instruction is executed. In either case, A is deleted from the stack. For binary coding details, refer to [Table B-4](#) on page B-4.

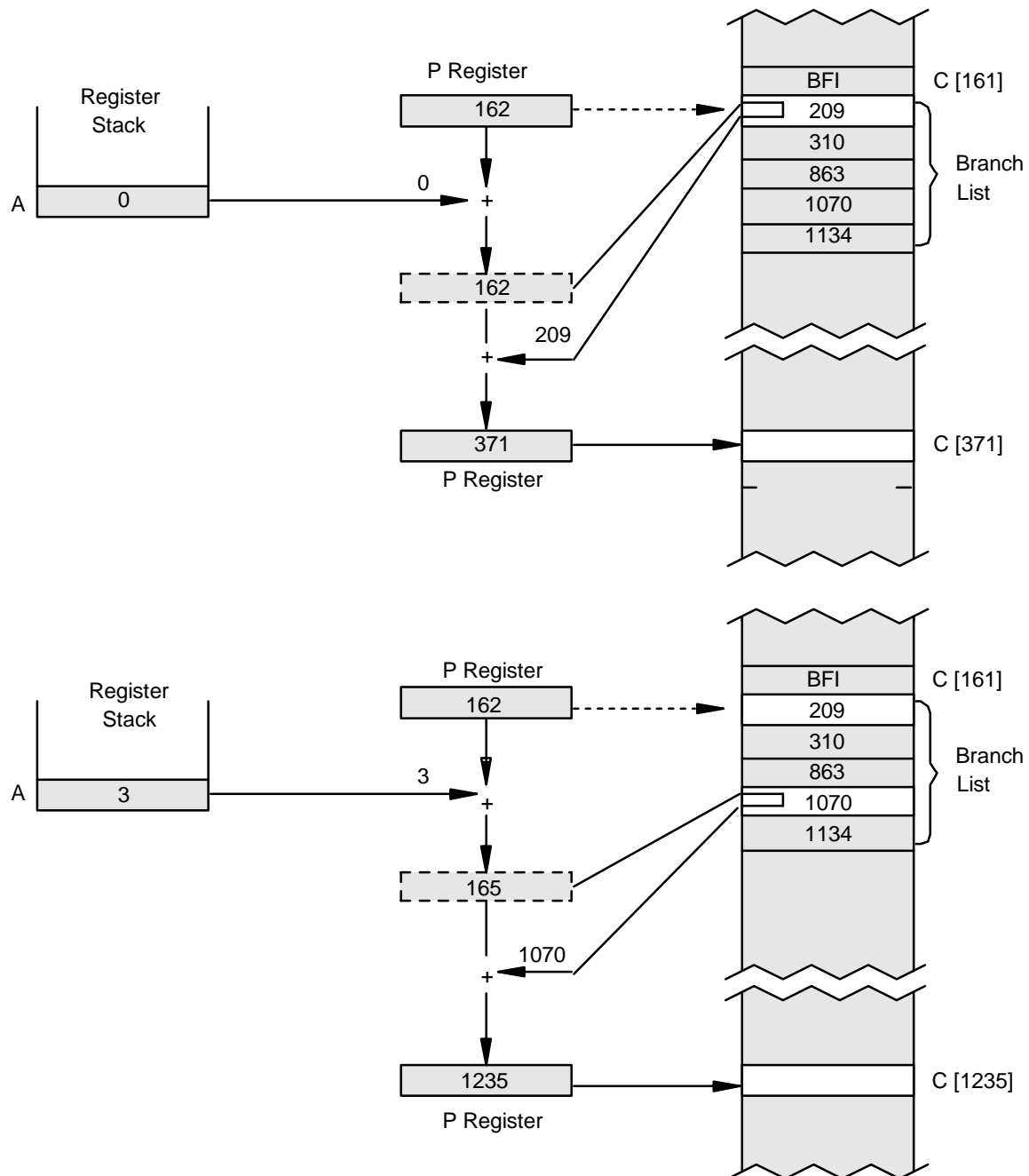
BEQL (-12---). Branch if CC Is Equal. If the Condition Code in the ENV register is CCE (N = 0, Z = 1), a direct or indirect branch is taken (depending on the “i” field of the instruction). If the condition is not met, the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

BFI (000030). Branch Forward Indirect. The instruction expects an offset from the current P register setting to be contained in A. An indirect branch is then made

through the location specified by  $P + A$ . [Figure 11-8](#) illustrates the action of the BFI instruction.

Both examples of BFI in [Figure 11-8](#) use the same branch list. In the first example,  $A$  is 0, so the first item in the list is used as the offset from  $P$ . In the second example,  $A$  is 3, so the fourth item in the list is used.

**Figure 11-8. Two Examples of Branch Forward Indirect (BFI)**



VST326.vsd

**BGEQ (-13---).** Branch if CC Is Greater or Equal. If the Condition Code in the ENV register is CCG or CCE ( $N = 0$ ), a direct or indirect branch is taken (depending on the “i” field of the instruction). If the condition is not met, the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

**BGTR (-11---).** Branch if CC Is Greater. If the Condition Code in the ENV register is CCG ( $N = 0, Z = 0$ ), a direct or indirect branch is taken (depending on the “i” field of the instruction). If the condition is not met, the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

**BIC (-100--).** Branch if Carry. If the Carry bit (K) in the Environment Register is set ( $K = 1$ ), a direct or indirect branch is taken (depending on the “i” field of the instruction). If the condition is not met, the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

**BLEQ (-16---).** Branch if CC Is Less or Equal. If the Condition Code in the ENV register is CCL or CCE ( $N = 1$  or  $Z = 1$ ), a direct or indirect branch is taken (depending on the “i” field of the instruction). If the condition is not met, the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

**BLSS (-14---).** Branch if CC Is Less. If the Condition Code in the ENV register is CCL ( $N = 1$ ), a direct or indirect branch is taken (depending on the “i” field of the instruction). If the condition is not met, the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

**BNEQ (-15---).** Branch if CC Is Not Equal. If the Condition Code in the ENV register is not CCE ( $Z = 0$ ), a direct or indirect branch is taken (depending on the “i” field of the instruction). If the condition is not met, the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

**BNOC (-17---).** Branch if No Carry. If the Carry bit (K) in the ENV register is not set ( $K = 0$ ), a direct or indirect branch is taken (depending on the “i” field of the instruction). If the condition is not met, the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

**BNOV (-164--).** Branch if No Overflow. If the Overflow bit (V) in the ENV register is not set ( $V = 0$ ), a direct or indirect branch is taken (depending on the “i” field of the instruction). If the condition is not met, the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

**BOX (-1-4--).** Branch on Index Less Than A and Increment X. If the index register as specified by the “x” field of the instruction is less than A, that index register is incremented and a direct or indirect branch is taken (depending on the “i” field of the instruction). If x is greater than or equal to A, A is deleted from the stack and the next instruction is executed. For binary coding details, refer to [Table B-4](#) on page B-4.

**BPT (000451).** Instruction Breakpoint Trap. This instruction, although necessarily nonprivileged, can be used effectively only by system software (Debug). The instruction assumes that Debug has inserted the BPT instruction at some user-specified point in the code and has saved the instruction that formerly occupied that location in the breakpoint table in the system data segment. When the code containing

the BPT instruction is executed, a Debug trap occurs when the BPT instruction is reached. When Debug has completed processing the breakpoint event, it can resume process execution at that interrupted point. Debug can leave the BPT instruction in place or it can have replaced it with the original contents of that location. In the former case, execution resumes in a special, temporary mode that fetches the missing instruction word from the system's breakpoint table rather than from C[P].

BSUB (-174--). Branch to Subprocedure. S is incremented by 1 and the return address (P) is saved in that location. Then a direct or indirect unconditional branch is taken (depending on the "i" field of the instruction). For binary coding details, see [Table B-4](#) on page B-4.

BTST (000007). Byte Test A. The Condition Code is set on the value of the test byte in bits 8:15 of A; CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character. A is deleted after the test.

BUN (-104--). Branch Unconditionally. A direct or indirect unconditional branch is taken (depending on the "i" field of the instruction). For binary coding details, refer to [Table B-4](#) on page B-4.

CAQ (000262). Convert ASCII to Quad. A string of 7-bit ASCII-coded digits in memory, defined by a starting byte address in B and a byte count in A, is converted to a binary-coded quadrupleword integer. The quadrupleword result is pushed onto the stack. If a nondigit ASCII code is encountered, only the preceding digits are converted and CCG indicates that only part of the string was converted; CCE indicates that the entire string was converted. Overflow is set if the result is greater than  $2^{63}-1$  or less than  $-2^{63}$ . If overflow is set, the value in DCBA is undefined.

CAQV (000261). Convert ASCII to Quad With Initial Value. A string of ASCII-coded digits in memory, defined by a starting byte address in F and a byte count in E, is converted to a binary-coded quadrupleword integer in DCBA. DCBA contains an initial value (greater than or equal to zero) that is multiplied by 10, providing a high-order value to which the converted value is added to produce the result in DCBA. If a nondigit ASCII code is encountered, only the preceding digits are converted and CCG indicates that only part of the string was converted; CCE indicates that the entire string was converted. Overflow is set if the result is greater than  $2^{63}-1$  or less than  $-2^{63}$ . If overflow is set, the value in DCBA is undefined.

CCE (000016). Set Condition Code to Equal. A Condition Code of CCE (N = 0 and Z = 1) is set into the ENV register.

CCG (000017). Set Condition Code to Greater. A Condition Code of CCG (N = 0 and Z = 0) is set into the ENV register.

CCL (000015). Set Condition Code to Less. A Condition Code of CCL (N = 1 and Z = 0) is set into the ENV register.

CDE (000334). Convert Double to Extended. The doubleword signed integer in BA is converted to an extended floating-point quantity. BA is deleted, and the four-word result is pushed onto the stack.



CDF (000306). Convert Double to Floating. The doubleword signed integer in BA is converted to a floating-point quantity in BA, with truncation if the result exceeds 23 significant bits.

CDFR (000326). Convert Double to Floating, Rounded. The doubleword signed integer in BA is converted to a floating-point quantity in BA, with rounding if the result exceeds 23 significant bits.

CDG (000366). Count Duplicate Words. Beginning at the address (in the current data segment) specified in register C, and for a maximum count of words specified in register B, this instruction counts the number of duplicate words in the buffer. Register A is incremented on each duplicate found and must contain an initial value. After execution, A contains the original A value plus the number of duplicate words, B contains a count of the words left in the buffer (zero if empty), and C contains the address of the first word that did not match its predecessor (or the word after the last word in the buffer). The comparison actually starts with the words specified by C and C-1. This instruction is intended to be used in conjunction with MNGG.

CDI (000307). Convert Double to Integer. The doubleword integer in BA is converted to a single-word integer by copying the contents of A into B and deleting A. Overflow is set if the doubleword quantity is greater than 32767 or less than -32768.

CDQ (000265). Convert Double to Quad. The doubleword integer in BA is extended to a quadrupleword quantity, filling the most significant two words with zeros if B is positive or ones if B is negative. BA is deleted, and the quadrupleword result is pushed onto the stack.

CDX (000356). Count Duplicate Words, Extended. Beginning at the 32-bit even-byte address specified in DC, and for a maximum count of words specified in B, this instruction counts the number of duplicate words in the buffer. A is incremented on each duplicate found and must contain an initial value. After execution, A contains the original A value plus the number of duplicate words, B contains a count of the words left in the buffer (zero if empty), and DC contains the extended address of the first word that did not match its predecessor (or the word after the last word in the buffer). The comparison actually starts with the words specified by DC and DC-2. This instruction is intended to be used in conjunction with MNDX.

CED (000314). Convert Extended to Double. The extended floating-point quantity in DCBA is converted to a doubleword signed integer. BA is deleted, and the doubleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than  $2^{31}-1$  or less than  $-2^{31}$ . Condition Code is set on the result.

CEDR (000315). Convert Extended to Double, Rounded. The extended floating-point quantity in DCBA is converted to a doubleword signed integer, with rounding according to the contents of the most significant fractional bit. BA is deleted, and the doubleword result is pushed onto the stack. Overflow is set if the value of the operand was greater than  $2^{31}-1$  or less than  $-2^{31}$ . Condition Code is set on the result.

CEF (000276). Convert Extended to Floating. The four-word floating-point quantity in DCBA is converted to a two-word floating-point quantity. DCBA is deleted, and the two-word result is pushed onto the stack.

CEFR (000277). Convert Extended to Floating, Rounded. The four-word floating-point quantity in DCBA is converted to a two-word floating-point quantity. The new quantity is rounded according to the contents of truncated bit 7 of C. DCBA is deleted, and the two-word result is pushed onto the stack.

CEI (000337). Convert Extended to Integer. The extended floating-point quantity in DCBA is converted to a single-word signed integer. CBA is deleted, and the single-word result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CEIR (000316). Convert Extended to Integer, Rounded. The extended floating-point quantity in DCBA is converted to a single-word signed quantity, with rounding according to the contents of the most significant fractional bit. CBA is deleted, and the single-word result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CEQ (000322). Convert Extended to Quadruple. The extended floating-point quantity in DCBA is converted to a quadrupleword integer in DCBA. Overflow is set if the value of the operand was greater than  $2^{63}-1$  or less than  $-2^{63}$ . Condition Code is set on the result.

CEQR (000323). Convert Extended to Quadruple, Rounded. The extended floating-point quantity in DCBA is converted to a quadrupleword integer in DCBA, with rounding according to the contents of the most significant fractional bit. Overflow is set if the value of the operand was greater than  $2^{63}-1$  or less than  $-2^{63}$ . Condition Code is set on the result.

CFD (000312). Convert Floating to Double. The floating-point quantity in BA is converted to a doubleword signed integer in BA. Overflow is set if the value of the operand was greater than  $2^{31}-1$  or less than  $-2^{31}$ . Condition Code is set on the result.

CFDR (000313). Convert Floating to Double, Rounded. The floating-point quantity in BA is converted to a doubleword signed integer in BA, with rounding according to the contents of the most significant fractional bit. Overflow is set if the value of the operand was greater than  $2^{31}-1$  or less than  $-2^{31}$ . Condition Code is set on the result.

CFE (000325). Convert Floating to Extended. The floating-point quantity in BA is converted to an extended floating-point quantity. BA is deleted, and the four-word result is pushed onto the stack.

CFI (000311). Convert Floating to Integer. The floating-point quantity in BA is converted to a single-word signed integer. A is deleted, and the single-word result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CFIR (000310). Convert Floating to Integer, Rounded. The floating-point quantity in BA is converted to a single-word signed integer, with rounding according to the contents of the most significant fractional bit. A is deleted, and the single-word result is pushed onto the stack. Overflow is set if the value of the operand was greater than 32767 or less than -32768. Condition Code is set on the result.

CFQ (000320). Convert Floating to Quadruple. The floating-point quantity in BA is converted to a quadrupleword integer in DCBA. Overflow is set if the value of the operand was greater than  $2^{63}-1$  or less than  $-2^{63}$ . Condition Code is set on the result.

CFQR (000321). Convert Floating to Quadruple, Rounded. The floating-point quantity in BA is converted to a quadrupleword integer in DCBA, with rounding according to the contents of the most significant fractional bit. Overflow is set if the value of the operand was greater than  $2^{63}-1$  or less than  $-2^{63}$ . Condition Code is set on the result.

CID (000327). Convert Integer to Double. The single-word integer in A is extended to a doubleword quantity on the top of the register stack. A is copied into H, and then A is filled with zeros if A was positive, or ones if A was negative; the register pointer is incremented to give the result in BA.

CIE (000332). Convert Integer to Extended. The signed integer in A is converted to an extended floating-point quantity. A is deleted, and the four-word result is pushed onto the stack.

CIF (000331). Convert Integer to Floating. The signed integer in A is converted to a floating-point quantity. A is deleted, and the two-word result is pushed onto the stack.

CIQ (000266). Convert Integer to Quad. The single-word integer in A is extended to a quadrupleword quantity, filling the most significant three words with zeros if A was positive or ones if A was negative. A is deleted, and the quadrupleword result is pushed onto the stack.

CLQ (000267). Convert Logical to Quad. The single-word logical quantity in A is extended to a quadrupleword quantity, filling the most significant three words with zeros. A is deleted, and the quadrupleword result is pushed onto the stack.

CMBX (000422). Compare Bytes Extended. This instruction compares one area of extended memory with another, a byte at a time, until the bytes are not equal or until a specified number of comparisons have been made. This instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, and ED to contain a 32-bit destination byte address. The instruction fetches the contents of the source and destination addresses, compares them, increments the addresses by 1, and decrements the byte count in A until either  $A = 0$  or a noncomparison is reached. If termination is due to a noncomparison, either CCG indicates that the byte at ED is greater than the byte at CB, or CCL indicates that the byte at ED is less than the byte at CB; A indicates the count of bytes left to compare. If termination is due to the count running out, CCE indicates that all bytes compared exactly; ED and CB point to the bytes after the last ones compared, and A is 0.

CMPI (001---). Compare A With Immediate Operand. The Condition Code is set as a result of the 16-bit integer comparison of A and the immediate operand. A is then deleted from the stack. Examples of the use of immediate operands are shown in [Figure 11-3](#) on page 11-4.

COMB (1262--). Compare Bytes. This instruction compares one area of memory with another, a byte at a time, until the bytes are not equal or until a specified number of comparisons have been made. It expects A to contain a byte count, B to contain a

source byte address, and C to contain a destination byte address. The source and destination segments to be used are specified by the “S” and “D” fields of the instruction and by the DS, CS, and LS, bits of the ENV register. If the source address is specified as current code segment, the byte address is taken to be in the same 64K half of the code space as the current P register value. If the source address is specified as latest user code segment, the byte address is taken to be in the lower 64K-byte half of that segment. The “RL” field determines whether the source and destination addresses will be incremented (“RL” = 0) or decremented (“RL” = 1) after each comparison. The “RP” field is the value to which RP will be set upon instruction termination. The instruction fetches the contents of source and destination addresses, compares them, increments or decrements the addresses by 1 according to the “RL” field, and decrements the byte count in A until either A = 0 or a noncomparison is reached. If termination is due to a noncomparison, either CCG indicates that the byte at C is greater than the byte at B, or CCL indicates that the byte at C is less than the byte at B; A indicates the number of bytes left to compare. If termination is due to the count running out, CCE indicates that all bytes compared exactly, and C and B will point to the next locations not compared. For binary coding details, refer to [Table B-3](#) on page B-3. [Figure 11-7](#) on page 11-9 provides a comparison of ascending and descending directions.

COMW (0262--). Compare Words. This instruction compares one area of memory with another, a word at a time, until a noncomparison occurs or until a specified number of comparisons have been made. The words being compared are treated as unsigned quantities. COMW expects A to contain a word count, B to contain a source word address, and C to contain a destination word address. The source and destination segments to be used are specified by the “S” and “D” fields of the instruction and by the DS, CS, and LS, bits of the ENV register. The “RL” field determines whether the source and destination addresses will be incremented (“RL” = 0) or decremented (“RL” = 1) after each comparison. The “RP” field is the value to which RP will be set upon instruction termination. The instruction fetches the contents of source and destination addresses, compares them, increments or decrements the addresses by 1 according to the “RL” field, and decrements the word count in A until either A = 0 or a noncomparison is reached. If termination is due to a noncomparison, CC indicates the results of the compare; CCE indicates that A has gone to zero. For binary coding details, refer to [Table B-3](#) on page B-3. [Figure 11-7](#) on page 11-9 provides a comparison of ascending and descending directions.

CQA (000260). Convert Quad to ASCII. The absolute value of the binary-coded quadrupleword integer in FEDC is converted to a string of ASCII-coded digits (decimal base), and the resulting string is stored in the memory space defined by a starting byte address in B and a byte count in A. If the conversion results in a truncation of leading digits, overflow is set. Condition Code is set on the original value.

CQD (000247). Convert Quad to Double. The four-word value in DCBA is converted to a doubleword by extracting the least significant two words. DCBA is deleted, and the doubleword result is pushed onto the stack. Overflow is set if the operand was greater than  $2^{31}-1$  or less than  $-2^{31}$ .

CQE (000336). Convert Quadruple to Extended. The quadrupleword signed integer in DCBA is converted to an extended floating-point quantity in DCBA, with truncation if the result exceeds 55 significant bits.

CQER (000335). Convert Quadruple to Extended, Rounded. The quadrupleword signed integer in DCBA is converted to an extended floating-point quantity in DCBA, with rounding if the result exceeds 55 significant bits.

CQF (000324). Convert Quadruple to Floating. The quadrupleword signed integer in DCBA is converted to a floating-point quantity, with truncation if the result exceeds 23 significant bits. DCBA is deleted, and the two-word result is pushed onto the stack.

CQFR (000330). Convert Quadruple to Floating, Rounded. The quadrupleword signed integer in DCBA is converted to a floating-point quantity, with rounding if the result exceeds 23 significant bits. DCBA is deleted, and the two-word result is pushed onto the stack.

CQI (000264). Convert Quad to Integer. The four-word value in DCBA is converted to an integer by extracting the least significant word. DCBA is deleted, and the integer result is pushed onto the stack. Overflow is set if the operand was greater than 32767 or less than -32768.

CQL (000246). Convert Quad to Logical. The four-word value in DCBA is converted to a logical value by extracting the least significant word. DCBA is deleted, and the integer result is pushed onto the stack. Overflow is set if the operand was greater than 65535.

DADD (000220). Double Add DC to BA. The two doubleword integers contained in DC and BA are added in doubleword integer form. Both operands are then deleted, and the doubleword result is pushed onto the stack. Overflow is set if the result is greater than  $2^{31}-1$  or less than  $-2^{31}$ . Carry can be set, and Condition Code is set on the result.

DALS (1302--). Double Arithmetic Left Shift. If the shift count field is zero, the doubleword contained in CB is shifted left by the dynamic count contained in A. A is then deleted from the stack. However, if the shift count field is not zero, BA is shifted left by that number. The sign bit is preserved only in nonaccelerated mode; in accelerated mode, DALS is treated as DLLS (Double Logical Left Shift). On doubleword shifts, dynamic shift counts greater than 255 or less than 0 give undefined results. Condition Code is set. Refer to [Figure 11-4](#) on page 11-5 for a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

DARS (1303--). Double Arithmetic Right Shift. If the shift count field is zero, the doubleword contained in CB is shifted right, propagating the sign bit, by the dynamic count contained in A. A is then deleted from the stack. However, if the shift count field is not zero, BA is shifted right, propagating the sign bit, by that number. On doubleword shifts, dynamic shift counts greater than 255 or less than 0 give undefined results. Condition Code is set. Refer to [Figure 11-4](#) on page 11-5 for a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

DCMP (000225). Double Compare DC With BA. The Condition Code in the ENV register is set as a result of the doubleword integer comparison of DC and BA. Both operands are then deleted from the stack.

DDIV (000223). Double Divide DC by BA. The doubleword integer contained in DC is divided in doubleword integer form by the doubleword integer in BA. Both operands are then deleted, and the result is pushed onto the stack. Overflow is set if the result is greater than  $2^{31}-1$  or less than  $-2^{31}$ , or if the divisor (BA) is zero. Carry can be set, and Condition Code is set on the result.

DDUP (000006). Double Duplicate BA in DC. The doubleword in the top two registers of the stack is duplicated by pushing a copy of it onto the register stack. Condition Code is set.

DFG (000367). Deposit Field in Memory. Using the mask bits in register B, this instruction deposits the bits in register C into the location specified by the 16-bit address in A. A, B, and C are then deleted. (See the DPF description for further details on this operation.) DFG accesses the current data segment. Condition Code is set.

DFS (000357). Deposit Field Into System Data. Using the mask bits in register B, this instruction deposits the bits in register C into the location specified by the 16-bit address in A. A, B, and C are then deleted. (See the DPF description for further details on this operation.) The destination is in the system data segment. A, B, and C are then deleted. Condition Code is set.

DFX (000416). Deposit Field Extended. Using the mask bits in register C, this instruction deposits the bits in register D into the memory location specified by the 32-bit even-byte address in registers B and A. All four words are then deleted from the register stack. (See the DPF description for further details on this operation.) Condition Code is set.

DISP (000073). Dispatch. This instruction sets bit 15 of INTA and also sets  $V_{i.<15>}$  in the system interrupt vector (SIV) table entry for the Dispatcher interrupt. If bit 15 of MASK is set, a Dispatcher interrupt occurs immediately following this instruction (provided there are no interrupts of higher priority pending). Control is then transferred to the operating system Dispatcher whose location is pointed to by the SIV table entry. This is a privileged instruction.

DLLS (1300--). Double Logical Left Shift. If the shift count field is zero, the doubleword contained in CB is shifted left by the dynamic count contained in A. A is then deleted from the stack. However, if the shift count field is not zero, BA is shifted left by that number. On doubleword shifts, dynamic shift counts greater than 255 or less than 0 give undefined results. Condition Code is set. Refer to [Figure 11-4](#) on page 11-5 for a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

DLRS (1301--). Double Logical Right Shift. If the shift count field is zero, the doubleword contained in CB is shifted right by the dynamic count contained in A. A is then deleted from the stack. However, if the shift count field is not zero, BA is shifted right by that number. On doubleword shifts, dynamic shift counts greater than 255 or



less than 0 give undefined results. Condition Code is set. Refer to [Figure 11-4](#) on page 11-5 for a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

**DMPY (000222).** Double Multiply DC by BA. The doubleword integer contained in DC is multiplied in doubleword integer form by the doubleword integer in BA. Both operands are then deleted, and the result is pushed onto the stack. Overflow is set if the result is greater than  $2^{31}-1$  or less than  $-2^{31}$ . Carry can be set, and Condition Code is set on the result.

**DNEG (000224).** Double Negate BA. The doubleword integer contained in BA is replaced with its two's complement. Overflow is set if the original operand was  $-2^{31}$ . Carry can be set, and Condition Code is set on the result.

**DPCL (000032).** Dynamic Procedure Call. Control is transferred to a procedure that is dynamically specifiable in the register stack (register A). The specified procedure can be in any of the four code spaces (UC, SC, UL, and SL). The format of the word in register A for specifying the target procedure is the same as that for a XEP table entry (see [Figure 6-36](#)). DPCL first stores a three-word stack marker, consisting of the current P, ENV, and L, on the top of the stack. (This copy of ENV includes the caller's space ID index in bits 11:15.) The address space mappings of relative segments 2 and 3 (current code segment and latest user code segment) are updated. Then, if the caller is not privileged, the PEP number is checked to see if the call is legal. If the call is not legal, an instruction failure trap occurs. If the caller is privileged, this check is not made. L and S are set to  $S + 3$  to point to the base of a new local data area. The final value of S is then checked for a value greater than 32767; if it is, a stack overflow trap occurs. Next, if the call is to a callable system procedure, the PRIV bit in the ENV register is set. CS and LS of ENV are set according to the corresponding bits of A (0 and 1, respectively). Finally, P is set from the PEP entry, transferring control to the target procedure.

**DPF (000014).** Deposit Field in A. This instruction combines the words contained in registers A and C of the stack as a function of a mask word contained in register B of the stack. A logical OR operation is performed on the logical AND of B and C and the logical AND of not B and A, so that all bits in C corresponding to ones in B are deposited into corresponding bits in A. The original three words are deleted from the stack and the result pushed onto the stack. Condition Code is set. An example of this operation is shown in [Figure 11-9](#) on page 11-22.

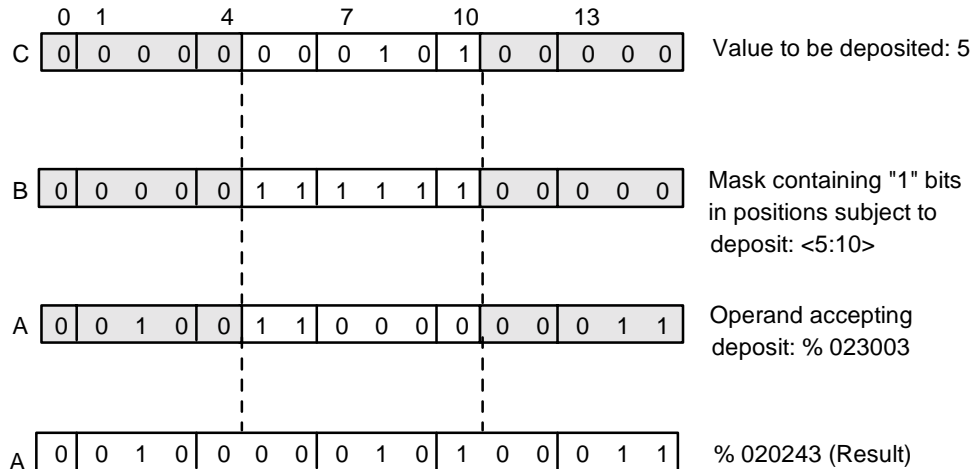
This example illustrates depositing a field using the DPF instruction. The principle also applies to the DFG, DFS, and DFX instructions. In all three cases, the source word is in C and the mask is in B. For DPF, the unmasked part of the source word is deposited into the existing contents of A. For DFG and DFS, the unmasked part of the source word is deposited into the memory cell addressed by A, located in the current user data segment or in the system data segment, respectively. (Register assignments are different for DFX, which uses B and A for an extended address.)

**Figure 11-9. Example of Depositing a Field Using the DPF Instruction**

INT i : % 023003

i.<5:10> := 5;

Values in register stack to do the above:



VST327.vsd

DSUB (000221). Double Subtract BA From DC. The doubleword integer contained in BA is subtracted in doubleword integer form from the doubleword integer in DC. Both operands are then deleted, and the result is pushed onto the stack. Overflow is set if the result is greater than  $2^{31}-1$  or less than  $-2^{31}$ . Carry can be set (meaning no borrow), and Condition Code is set on the result.

DTST (000031). Double Test BA. The Condition Code is set according to the contents of the doubleword contained in BA.

DXCH (000005). Double Exchange BA With DC. The doubleword contained in DC is interchanged with the doubleword contained in BA. Condition Code is set on the result in BA.

EADD (000300). Extended Add. The extended floating-point quantities in HGFE and DCBA are added in extended floating-point form. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. (For the range of extended floating-point numbers, refer to [Section 3, TNS Data Formats and Number Representations](#).) Condition Code is set on the result.

ECMP (000305). Extended Compare. The Condition Code is set according to the comparison of HGFE (operand 1) with DCBA (operand 2). Both operands are then deleted from the stack.

EDIV (000303). Extended Divide. The extended floating-point quantity in HGFE is divided in extended floating-point form by the extended floating-point quantity in DCBA. Both operands are deleted and the result is pushed onto the stack. Overflow is set if



the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

EMPY (000302). Extended Multiply. The extended floating-point quantities in HGFE and DCBA are multiplied in extended floating-point form. Both operands are deleted, and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

ENEG (000304). Extended Negate. The extended floating-point quantity in DCBA (if not zero) is negated. The sign of DCBA is reversed from positive to negative or negative to positive. Overflow is cleared, and the Condition Code reflects the final state of the sign.

ESUB (000301). Extended Subtract. The extended floating-point quantity in HGFE is negated, and then HGFE and DCBA are added in extended floating-point form. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of extended floating-point numbers. Condition Code is set on the result.

EXCH (000004). Exchange A and B. A and B of the register stack are interchanged. Condition Code is set on the result in A.

EXIT (125---). Exit From Procedure. This instruction is used to return from a procedure called by a PCAL, XCAL, or DPCL instruction. EXIT assumes L-2:L to contain a standard three-word stack marker consisting of P, ENV, and L. (This copy of ENV includes the caller's space ID index in bits 11:15.) The address space mappings of relative segments 2 and 3 (current code segment and latest user code segment) are updated. Then S is moved below the current stack marker and any parameters by setting it with the "S decrement" value subtracted from the current L register setting. P is set to the return P value contained in L[-2] of the current stack marker. The caller's ENV register value is set as follows: the mode (privileged or nonprivileged) and data area are reinstated to the lesser of the caller's and the current settings (so a privileged calling process can be made nonprivileged on the return, but not vice versa); the calling process's CS (code space), LS (library space), T (traps), V (overflow), and K (carry) bits are reinstated from L[-1]; Z and N (Condition Code) and RP are set to those of the current procedure. L is moved back to the preceding stack marker, thereby reinstating the preceding local data area, by setting L with the contents of the L[0] of the current stack marker. If bit 0 of the restored ENV is 1, a Debug breakpoint trap occurs. If the V and T bits of the restored ENV are both 1, an arithmetic overflow trap occurs.

FADD (000270). Floating-Point Add. The floating-point quantities in DC and BA are added in floating-point form. Both operands are deleted, and the two-word result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. (For the range of floating-point numbers, refer to [Section 3, TNS Data Formats and Number Representations](#).) Condition Code is set on the result.

FCMP (000275). Floating-Point Compare. The Condition Code is set according to the comparison of DC (operand 1) with BA (operand 2). (See [Table B-1](#) on page B-1 for

Condition Code settings; the “a” states apply for comparisons.) Both operands are then deleted from the stack.

**FDIV (000273).** Floating-Point Divide. The floating-point quantity in DC is divided in floating-point form by the floating-point quantity in BA. Both operands are deleted and the result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

**FMPY (000272).** Floating-Point Multiply. The floating-point quantities in DC and BA are multiplied in floating-point form. Both operands are deleted, and the result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

**FNEG (000274).** Floating-Point Negate. The floating-point quantity in BA (if not zero) is negated. The sign of BA is reversed from positive to negative or negative to positive, and the Condition Code reflects the final state of the sign (see [Table B-1](#) on page B-1).

**FSUB (000271).** Floating-Point Subtract. The floating-point quantity in BA is negated, and then DC and BA are added in floating-point form. Both operands are deleted, and the result is pushed onto the stack. Overflow is set if the result falls outside the range of floating-point numbers. Condition Code is set on the result.

**IADD (000210).** Integer Add A to B. A is added to B in integer (signed) form. A and B are then deleted from the stack and the sum is pushed onto the stack. Overflow is set if the result is greater than 32767 or less than –32768. Condition Code is set.

**ICMP (000215).** Integer Compare B With A. B is compared to A in integer (signed) form and the Condition Code set accordingly. A and B are then deleted from the stack.

**IDIV (000213).** Integer (Signed) Divide B by A. B is divided by A in integer (signed) form. A and B are deleted from the stack and the result pushed on. Overflow is set if the divisor is zero, or if the result is greater than 32767 or less than –32768. Condition Code is set.

**IDX1 (000344).** Calculate Index, 1 Dimension. For a one-dimensional array, IDX1 compares the subscript value in B against lower and upper bounds in a two-word table in the current code segment starting at the address specified in A. The element offset value is computed and stored in register R[7]. Condition Code is set based on the result in R[7]. If the subscript is out of bounds, overflow is set. BA is then deleted.

**IDX2 (000345).** Calculate Index, 2 Dimensions. For a two-dimensional array, IDX2 compares the subscript values in B and C against lower and upper bounds in a four-word table in the current code segment starting at the address in A. The element offset value is computed and stored in register R[7]. Condition Code is set based on the result in R[7]. If a subscript is out of bounds, overflow is set. CBA is then deleted.

**IDX3 (000346).** Calculate Index, 3 Dimensions. For a three-dimensional array, IDX3 compares the subscript values in B, C, and D against lower and upper bounds in a six-word table in the current code segment starting at the address in A. The element offset value is computed and stored in register R[7]. Condition Code is set based on

the result in R[7]. If any subscript is out of bounds, overflow is set. DCBA is then deleted.

IDXD (000317). Calculate Index, Data Space. For an n-dimensional array, IDXD compares the subscript values in n stack registers (B, C, D, and so on) against lower and upper bounds in a table in the current data segment ( $2n + 1$  words) specified by a starting address in A. (The first word of the table in memory is the number of dimensions; bit 0 is 1 if bounds checking is to be suppressed.) The element offset value is computed and stored in register R[7]. Condition Code is set based on the result in R[7]. If bounds checks are enabled and any subscript is out of bounds, overflow is set. All stack data used is deleted.

IDXP (000347). Calculate Index, Code Space. For an n-dimensional array, IDXP compares the subscript values in n stack registers (B, C, D, and so on) against lower and upper bounds in a table in the current code segment ( $2n + 1$  words) specified by a starting address in A. (The first word of the table in memory is the number of dimensions; bit 0 is 1 if bounds checking is to be suppressed.) The element offset value is computed and stored in register R[7]. Condition Code is set based on the result in R[7]. If bounds checks are enabled and any subscript is out of bounds, overflow is set. All stack data used is deleted.

IMPY (000212). Integer Multiply A Times B. B is multiplied by A in integer (signed) form. A and B are deleted from the stack and the result pushed on. Overflow is set if the result is greater than 32767 or less than -32768. Condition Code is set.

INEG (000214). Integer Negate A. The signed integer in A is converted to its two's-complement form. Overflow is set if the original operand was -32768. Carry is set, meaning no borrow. Condition Code is set.

ISUB (000211). Integer Subtract A From B. A is subtracted from B in integer (signed) form. A and B are deleted, and the difference is pushed onto the stack. Overflow is set if the result is greater than 32767 or less than -32768. Carry is set, meaning no borrow. Condition Code is set.

LADD (000200). Logical Add A to B. A and B are added as 16-bit positive (unsigned) integers. A and B are then deleted from the stack and the result pushed on. Carry is set if the addition overflows bit 0. Condition Code is set.

LADI (003---). Logical Add Immediate Operand to A. The immediate operand is pushed onto the stack, with the sign bit propagating into the high-order bits. Then A and B are added in 16-bit unsigned integer form. A and B are then both deleted from the stack and the result pushed on. Carry is set if the addition overflows bit 0. Condition Code is set. Examples of the use of immediate operands are shown in [Figure 11-3](#) on page 11-4.

LADR (-70---). Load G-Relative Address of Variable Into A. The G-relative address of the variable is pushed onto the stack. For binary coding details, refer to [Table B-1](#) on page B-1.

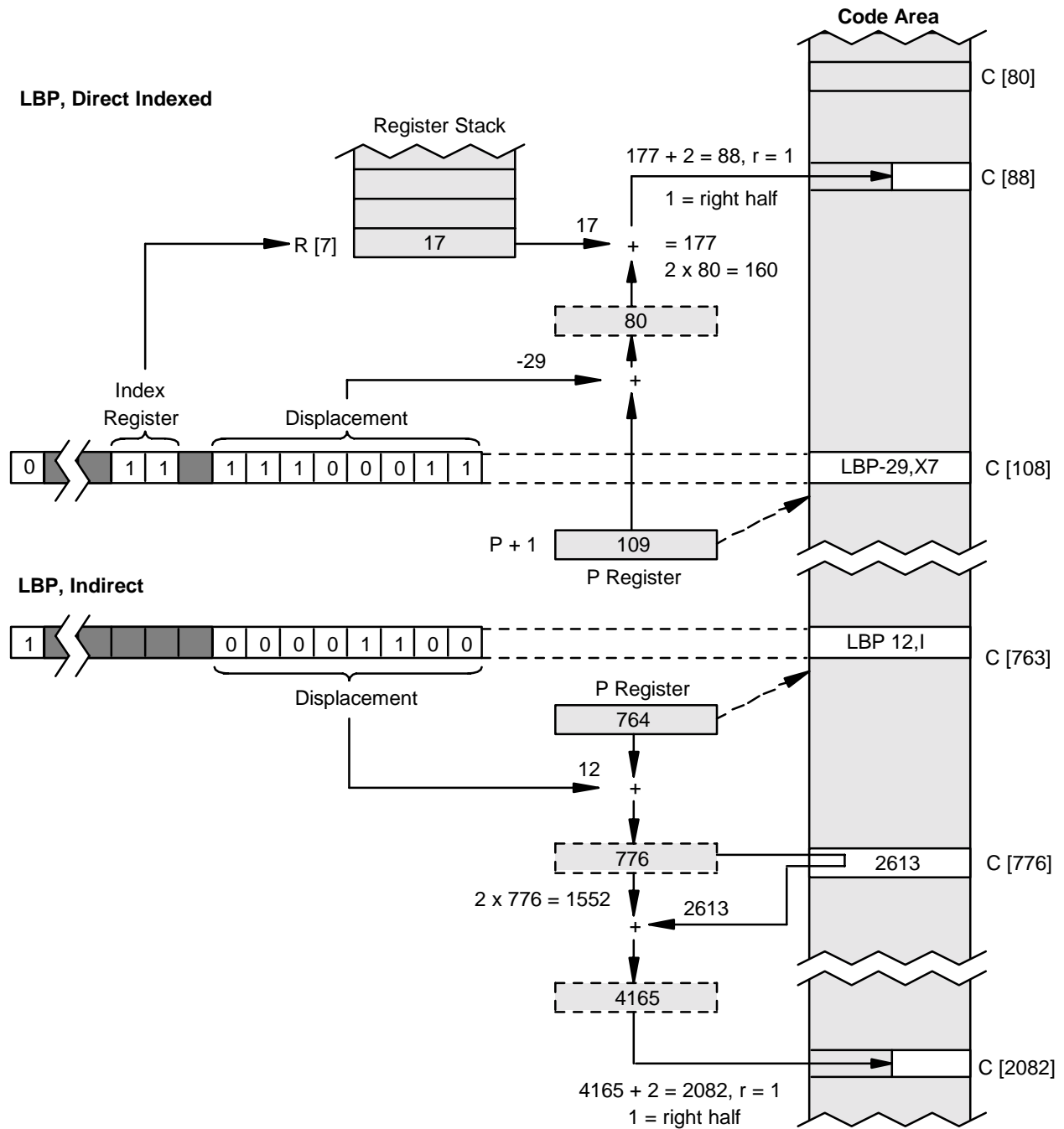
LAND (000010). Logical AND A With B. A and B are logically ANDed. The two words are deleted from the stack and the result pushed on. Condition Code is set. (For an example, see [Figure 11-5](#) on page 11-6.)

LBA (000364). Load Byte via A. The unsigned byte contained in the effective memory location pointed to by the byte address in A is loaded onto the stack, with zero extension, replacing the prior contents of A. LBA accesses the current data segment only. The Condition Code is set on the category of the loaded byte in A: CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character.

LBAS (000354). Load Byte via A From System. The unsigned byte contained in the effective memory location pointed to by the byte address in A is loaded onto the stack, with zero extension, replacing the prior contents of A. A refers to an address in the system data segment. The Condition Code is set on the category of the loaded byte in A: CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character.

LBP (-2-4--). Load Byte From Program (Current Code Segment) Into A. The contents of the P-relative byte address that is computed as a function of displacement (a signed 8-bit value), and optionally indexing and indirection, are pushed onto the register stack. The high-order byte is set to zero. The addressing range is half a segment. For direct addressing, if the P register currently indicates an address in the upper half of the code segment (bit 0 of P = 1), %100000 is added to the computed address, so that the address will always be relative to whichever half of the segment P currently indicates. For indirect addressing, the address is based on the indirect cell's half segment, not the half segment that contains the LBP instruction. The Condition Code is set on the category of the loaded byte in A: CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character. [Figure 11-10](#) on page 11-27 illustrates the addressing operations for the LBP instruction, assuming addresses in the first half of the code segment. For binary coding details, refer to [Table B-1](#) on page B-1.

The first example of LBP instruction addressing shown in [Figure 11-10](#) is direct, using a negative displacement value and an index value of 17. The second example is indirect and uses a positive displacement.

**Figure 11-10. Two Examples of Load Byte From Program (LBP)**

VST328.vsd

**LBX (000406).** Load Byte Extended. The unsigned byte in the memory location specified by the 32-bit byte address in registers B and A is loaded onto the register stack (bits 8 through 15 of A), after the address in BA is deleted. The left byte of A is set to zero. The Condition Code is set on the category of the loaded byte in A: CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character.

**LBXX (0256--, 0266--).** Load Byte Extended, Indexed. The unsigned byte contained in a computed memory location is loaded onto the stack, with zero extension, replacing the prior contents of A. The memory address is obtained as follows. The word displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value that is either G[0] (coded 0256--) or the current L register value (coded 0266--); the data word so indicated is assumed to be the first word of a two-word extended memory pointer. The index value in A is then added to the extended memory pointer to address the byte that is to be loaded. The Condition Code is set on the category of the loaded byte in A: CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character. For binary coding details, refer to [Table B-3](#) on page B-3.

**LCMP (000205).** Logical Compare B With A. B is logically compared to A and the Condition Code set accordingly. A and B are then deleted from the stack.

**LDA (000362).** Load Double via A. The doubleword contained in the effective memory locations starting at the location pointed to by the address in A is loaded into BA (after the address in A is deleted). LDA accesses the current data segment only. Condition Code is set.

**LDAS (000352).** Load Double via A From System. The doubleword contained in the effective memory locations starting at the location pointed to by the address in A is loaded into BA (after the address in A is deleted). A refers to an address in the system data segment. Condition Code is set.

**LDB (-50---).** Load A With Byte From Data Space. The contents of the effective memory location are loaded into bits 8:15 of A; the left byte is set to zero. (Refer to [Byte Addressing in the Data Segment](#) on page 6-22 for calculation of the effective address in byte addressing.) The Condition Code is set on the category of the loaded byte in A: CCL indicates ASCII numeric, CCE indicates ASCII alphabetic, and CCG indicates special ASCII character. For binary coding details, refer to [Table B-1](#) on page B-1.

**LDD (-60---).** Load Doubleword from Data Space Into BA. The doubleword integer contained in the effective memory location is pushed onto the stack. Condition Code is set. [Figure 11-2](#) on page 11-3 illustrates the addressing methods for doubleword instructions. For binary coding details, refer to [Table B-1](#) on page B-1.

**LDDX (000412).** Load Doubleword Extended. The doubleword starting at the memory location specified by the 32-bit even-byte address in registers B and A is loaded onto the register stack, replacing the prior contents of B and A. Condition Code is set.



LDI (100---). Load Immediate Operand Into A. The immediate operand is pushed onto the stack, with the sign bit propagating into the high-order bits. Condition Code is set. (For binary coding details, refer to [Table B-2](#) on page B-2.)

LDIV (000203). Logical Divide CB by A, leaving the remainder in B. The 32-bit positive (unsigned) integer in C and B is divided by the 16-bit positive integer in A. The divisor and dividend are deleted from the stack, the remainder is pushed onto the stack (B), and the quotient is pushed onto the stack (A). Overflow is set if the original C is greater than or equal to the original A. Condition Code is set, based on the quotient in A.

LDLI (005---). Load Left Immediate Operand Into Bits 0:7 of A. The immediate operand, shifted left eight places, is loaded into A, with the sign bits propagating into the low-order bits of A. Condition Code is set. (For binary coding details, refer to [Table B-2](#) on page B-2.)

LDRA (00013-). Load A From a Register. The contents of the register pointed to by the Register field of the instruction are pushed onto the stack. Condition Code is set. (For binary coding details, refer to [Table B-5](#) on page B-5.)

LDX (-3x---). Load Index Register From Data Space. The index register specified by the “x” field of the instruction is loaded with the contents of the effective memory address. Condition Code is set. [Figure 11-1](#) on page 11-2 shows the instruction word format for memory data reference instructions, such as LDX. For binary coding details, refer to [Table B-1](#) on page B-1.

LDXI (10----). Load Index Register With Immediate Operand. The index register specified by the “x” field of the instruction is loaded with the immediate operand, and the sign bit propagates into the high-order bits. Condition Code is set. (For binary coding details, refer to [Table B-2](#) on page B-2.)

LLS (0300--). Logical Left Shift. If the shift count field is zero, the (unsigned) word contained in B is shifted left by the dynamic count contained in A. A is then deleted from the stack. However, if the shift count field is not zero, A is shifted left by that number. Condition Code is set. Refer to [Figure 11-4](#) on page 11-5 for a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

LMPY (000202). Logical Multiply A Times B. A and B are multiplied as 16-bit positive (unsigned) integers. A and B are then replaced by the doubleword result, with the least significant half in A. Overflow is implicitly cleared. Condition Code is set.

LNEG (000204). Logical Negate A. The unsigned value in A is converted to its two's complement. Carry is set if the original value of A is zero. Condition Code is set.

LOAD (-40---). Load A From Data Space. The contents of the effective address memory location are pushed onto the stack. Condition Code is set. For binary coding details, refer to [Table B-1](#) on page B-1.

LOR (000011). Logical OR A With B. A and B are merged by a logical inclusive OR. A and B are deleted and the result pushed onto the stack. Condition Code is set. (For an example, see [Figure 11-5](#) on page 11-6.)

LQAS (000445). Load Quadrupleword via A From SG. The quadrupleword contained in the four memory locations starting at the location pointed to by the address in A is loaded into DCBA (after the address in A is deleted). The address in A refers to an address in the system data segment. Condition Code is set. This is a privileged instruction.

LQX (000414). Load Quadrupleword Extended. The quadrupleword starting at the memory location specified by the 32-bit even-byte address in registers B and A is loaded into registers DCBA of the register stack (after the address in BA is deleted). Condition Code is set.

LRS (0301--). Logical Right Shift. If the shift count field is zero, the (unsigned) word contained in B is shifted right by the dynamic count contained in A. A is then deleted from the stack. However, if the shift count field is not zero, A is shifted right by that number. On single-word shifts, shift counts greater than 31 or less than 0 give undefined results. Condition Code is set. Refer to [Figure 11-4](#) on page 11-5 for a comparison of logical (unsigned) shifts and arithmetic (signed) shifts.

LSUB (000201). Logical Subtract A From B. A is subtracted from B logically (unsigned). A and B are then deleted from the stack and the result pushed on. Carry is set if no borrow occurs; that is, A is less than or equal to B (on unsigned basis). Condition Code is set.

LWA (000360). Load Word via A. The word contained in the effective memory location pointed to by the address in A is loaded onto the stack, replacing the prior contents of A. LWA accesses the current data segment only. Condition Code is set.

LWAS (000350). Load Word via A From System. The word contained in the effective memory location pointed to by the address in A is loaded onto the stack, replacing the prior contents of A. A refers to an address in the system data segment. Condition Code is set.

LWP (-20---). Load Word from Program (Current Code Segment) Into A. The contents of the address that is computed as a function of displacement (a signed 8-bit value), and optionally indirection and indexing, are pushed onto the register stack. Condition Code is set on the loaded word. For binary coding details, refer to [Table B-1](#) on page B-1.

LWUC (000342). Load Word From User Code Space. A word in the latest user code segment, specified by a 16-bit address in A, is loaded onto the stack, replacing the prior contents of A. Condition Code is set.

LWX (000410). Load Word Extended. The word in the memory location specified by the 32-bit even-byte address in registers B and A is loaded into register A (after the address in BA is deleted). Condition Code is set.

LWXX (0254--, 0264--). Load Word Extended, Indexed. The word contained in a computed memory location is loaded onto the stack, replacing the prior contents of A. The memory address is obtained as follows. The word displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value, which is either G[0] (coded 0254--) or the current L register value (coded 0264--); the data word



so indicated is assumed to be the first word of a two-word even-byte extended memory pointer. The index value in A is sign-extended, then shifted left one bit position (multiplication by 2, because this instruction requires word addressing rather than byte addressing) and then added to the extended memory pointer to address the word that is to be loaded. Condition Code is set. For binary coding details, refer to [Table B-3](#) on page B-3.

**MBXR (000420).** Move Bytes Extended, Reverse. This instruction transfers a specified number of bytes from one area of extended memory to another, using reverse (decrementing) addresses. The instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, and ED to contain a 32-bit destination byte address. The move is made one byte at a time from the source to the destination. After each byte transfer, the addresses are decremented and A is decremented. If A is equal to zero, the instruction ends; otherwise the next byte is moved. All five words are deleted from the stack when the instruction ends.

**MBXX (000421).** Move Bytes Extended, and Checksum. This instruction transfers a specified number of bytes from one area of extended memory to another and computes a checksum value (byte exclusive “or”) after each byte is moved. The instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, ED to contain a 32-bit destination byte address, and F to contain the initial checksum value. The move is made one byte at a time from the source to the destination. After each byte transfer, the addresses are incremented, A is decremented, and the new checksum is entered in F. If A is equal to zero, the instruction ends; otherwise the next byte is moved. Five words are deleted from the register stack when the instruction ends, leaving the final checksum value in A.

**MNDX (000227).** Move Words While Not Duplicate, Extended. FE is assumed to contain a 32-bit even-byte destination address, and DC is assumed to contain a 32-bit even-byte source address. The MNDX instruction moves words from the source to the destination while the count value in register B is not zero and the source word is not equal to the word in A. The word in A is always the previous word moved. The instruction stops on the first duplicate word or on zero count. After execution, the word in A is deleted, so that A then contains the count, CB contains the source address, and ED contains the destination address.

**MNGG (000226).** Move Words While Not Duplicate. Register D is assumed to contain a destination address in the current data segment, and register C is assumed to contain a source address in the current data segment. The MNGG instruction moves words from the source to the destination while the count value in register B is not zero and the source word is not equal to the word in A. The word in A is always the previous word moved. The instruction stops on the first duplicate word or on zero count. After execution, the word in A is deleted, so that A then contains the count, B contains the source address, and C contains the destination address.

**MOND (000001).** Minus One Double. A doubleword negative one is pushed onto the top of the register stack (BA). Condition Code is set.

**MOVB (126---).** Move Bytes. This instruction transfers a specified number of bytes from one area of memory to another. The instruction expects A to contain a byte

count, B to contain the source byte address, and C to contain the destination byte address. The source and destination segments to be used are specified by the “S” and “D” fields of the instruction and by the DS, CS, and LS bits of the ENV register. The “RL” field of the instruction determines whether the source and destination addresses will be incremented (“RL” = 0) or decremented (“RL” = 1) after each move. The “RP” field of the instruction is the value to which RP is set upon instruction end. The move is made one byte at a time from the source to the destination. After each byte transfer, the addresses are decremented or incremented and A is decremented. If A is equal to zero, the instruction ends; otherwise the next byte is moved. If the source is specified as the current code segment and the P register currently indicates an address in the upper half of the code segment (bit 0 of P = 1), %100000 is added to the computed address, so that the source address is always relative to whichever half of the segment P currently indicates. If the source is specified as the latest user code segment, the 16-bit source address indexes the lower 64K bytes of that segment. For binary coding details, refer to [Table B-3](#) on page B-3. [Figure 11-7](#) on page 11-9 provides a comparison of ascending and descending directions.

MOVW (026---). Move Words. This instruction transfers a specified number of words from one area of memory to another. The instruction expects A to contain a word count, B to contain the source word address, and C to contain the destination word address. The source and destination segments to be used are specified by the “S” and “D” fields of the instruction and by the DS, CS, and LS bits of the ENV register. The “RL” field of the instruction determines whether the source and destination addresses will be incremented (“RL” = 0) or decremented (“RL” = 1) after each move. The “RP” field of the instruction is the value to which RP is set upon instruction end. The move is made one word at a time from the source to the destination. After each word transfer, the addresses are decremented or incremented and A is decremented. If A is equal to zero, the instruction ends; otherwise the next word is moved. For binary coding details, refer to [Table B-3](#) on page B-3. [Figure 11-7](#) on page 11-9 provides a comparison of ascending and descending directions.

MVBX (000417). Move Bytes Extended. This instruction transfers a specified number of bytes from one area of extended memory to another. The instruction expects A to contain a byte count, CB to contain a 32-bit source byte address, and ED to contain a 32-bit destination byte address. The move is made one byte at a time from the source to the destination. After each byte transfer, the addresses are incremented and A is decremented. If A is equal to zero, the instruction ends; otherwise the next byte is moved. All five words are deleted from the stack when the instruction ends.

NOP (000000). No Operation.

NOT (000013). One’s Complement A. The word contained in register A of the stack is converted to its one’s complement. Condition Code is set. (For an example, see [Figure 11-5](#) on page 11-6.)

NSAR (00012-). Nondestructive Store A Into a Register. The A register is stored in the register pointed to by the Register field of the instruction. (For binary coding details, refer to [Table B-5](#) on page B-5.)

NSTO (-34---). Nondestructive Store From A. The contents of the A register are stored into the effective address memory location. The register stack is not modified. For binary coding details, refer to [Table B-1](#) on page B-1.

ONED (000003). One Double. A doubleword 1 is pushed onto the top of the register stack (BA). Condition Code is set.

ORG (000045). OR to Memory. The word in B is logically ORed to a word in the current data segment that is specified by the 16-bit address in A. The result remains in the data segment location, and A and B are deleted from the stack. Condition Code is set.

ORLI (0040--). OR Left Immediate Operand With A. The 8-bit immediate operand is shifted left eight places and merged with A by a logical inclusive OR. The sign bit is not propagated but is actually part of the instruction opcode. Condition Code is set. (For binary coding details, refer to [Table B-2](#) on page B-2; for an example, see [Figure 11-6](#) on page 11-7.)

ORRI (0044--). OR Right Immediate Operand With A. The 8-bit immediate operand is merged with the A register by a logical inclusive OR. The sign bit is not propagated but is actually part of the instruction opcode. Condition Code is set. (For binary coding details, refer to [Table B-2](#) on page B-2; for an example, see [Figure 11-6](#) on page 11-7.)

ORS (000035). OR to SG Memory. The word in B is logically ORed to a word in the system data segment that is specified by the 16-bit address in A. The result remains in the system data location, and A and B are deleted from the stack. A refers to an address in the system data segment. Condition Code is set.

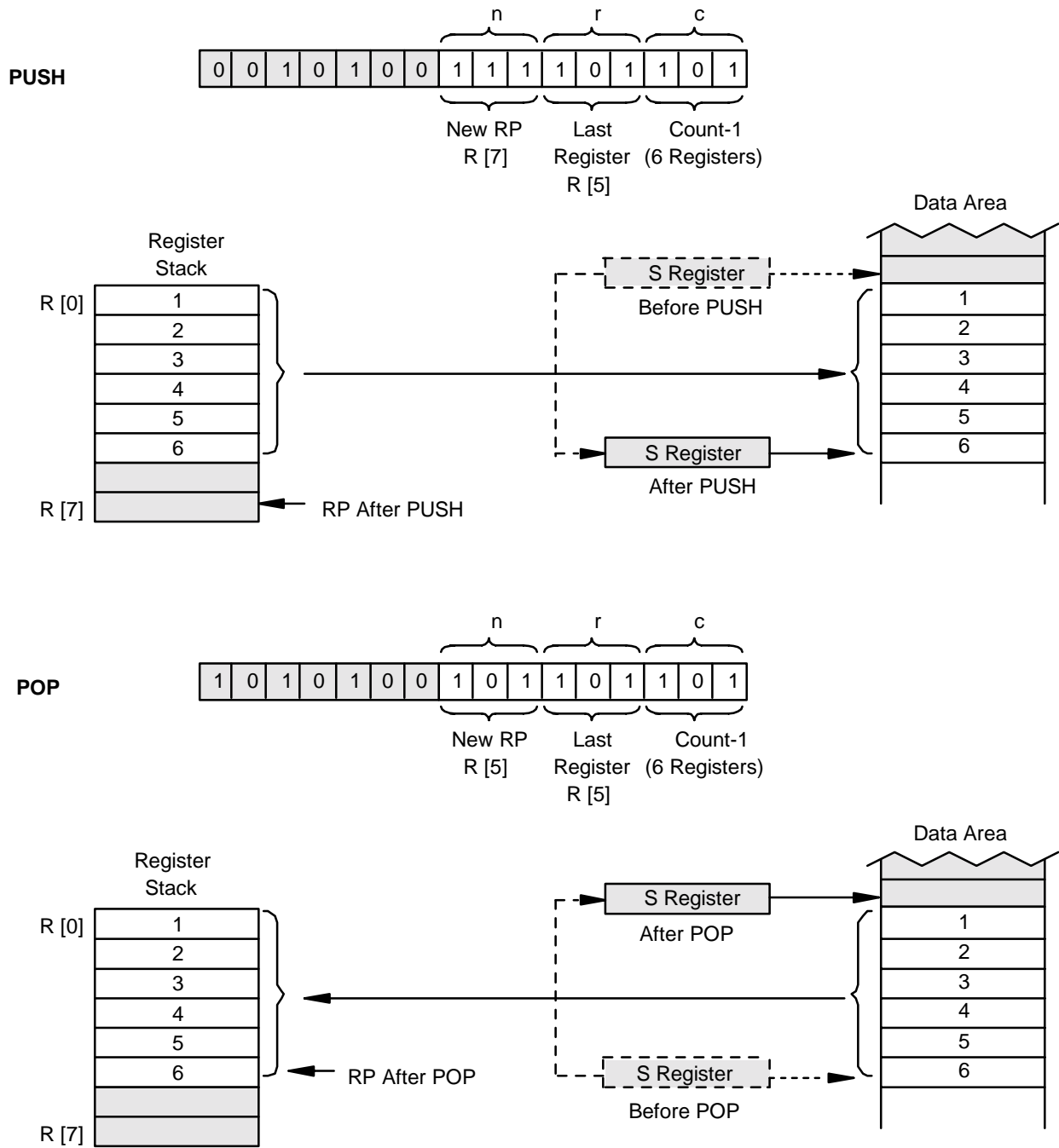
ORX (000047). OR to Extended Memory. The word in C is logically ORed to a word in memory that is specified by the 32-bit even-byte address in BA. The result remains in the memory location, and A, B, and C are deleted from the stack. Condition Code is set.

PCAL (027---). Procedure Call. Control is transferred to an instruction specified by an entry in the procedure entry point (PEP) table; the specific PEP entry is indicated by the PEP Number field of the instruction. First, a three-word stack marker, consisting of the current P, ENV, and L, is stored on the top of the current stack. (This copy of ENV includes the space ID index in bits 11:15; CC and RP are not preserved.) If the caller is not privileged, the PEP number is checked against PEP[0] and PEP[1] to see if the call is legal. If the call is not legal, an instruction failure trap occurs. (If the caller is privileged, no checks are made.) L and S are set to S + 3 to point to the base of a new local data area. The final value of S is then checked for a value greater than 32767; if it is, a stack overflow trap occurs. Finally, P is set from the PEP entry and control is transferred to the procedure.

POP (124nrc). Pop Data Space to Registers. This instruction loads the register stack with the top elements of the data stack (as indicated by the current S register setting). The “n” field of the instruction indicates the value RP will have following the instruction, the “r” field specifies the last register stack element to be loaded from memory, and the “c” field specifies the number of registers minus one that will be loaded. If the resultant value of S is greater than 32767, a stack overflow trap occurs. [Figure 11-11](#) on page 11-35 illustrates the bit fields and the action of the POP instruction. For binary coding details, refer to [Table B-3](#) on page B-3.

PUSH (024nrc). Push Registers to Data Space. This instruction transfers the contents of a specified number of elements in the register stack to the top of the data stack in memory. The “n” field of the instruction is the value to which RP will be set following the instruction, the “r” field specifies the last register stack element to be pushed, and the “c” field is the number of registers minus one that will be pushed to memory. Following the PUSH instruction, the S register points to the last element pushed onto the memory stack. If the resultant value of S is greater than 32767, a stack overflow trap occurs. [Figure 11-11](#) on page 11-35 illustrates the bit fields and the action of the PUSH instruction. For binary coding details, refer to [Table B-3](#) on page B-3.

In [Figure 11-11](#), the PUSH instruction transfers the contents of six registers of the register stack to the data stack in memory. The POP instruction does the reverse.

**Figure 11-11. Example of PUSH and POP Instructions**

VST329.vsd

**QADD (000240).** Quadruple Add. The two quadrupleword integers contained in HGFE and DCBA are added in quadrupleword integer form. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the result is greater than  $2^{63}-1$  or less than  $-2^{63}$ . Carry can be set, and Condition Code is set on the result.

**QCMP (000245).** Quadruple Compare. The Condition Code in the Environment register is set according to the quadruple integer comparison of HGFE (operand 1) and DCBA (operand 2). (See [Table B-1](#) on page B-1 for Condition Code settings; the “a” states apply for compares.) Both operands are then deleted from the stack.

**QDIV (000243).** Quadruple Divide. The quadrupleword integer contained in HGFE is divided in quadrupleword integer form by the quadrupleword integer in DCBA. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the divisor (DCBA) is zero. Condition Code is set.

**QDWN (00025-).** Quadruple Scale Down. The operand value in DCBA is divided by a specified power of ten (1, 2, 3, or 4), and the new value replaces the former contents of DCBA. Condition Code is set, and the Overflow bit is cleared. For binary coding details, refer to [Table B-6](#) on page B-7.

**QLD (00023-).** Quadruple Load. The quadrupleword operand contained in the effective memory location indicated by A plus 4 times the index value is fetched. A is deleted, and the fetched quadrupleword is pushed onto the stack. No indexing occurs for code 000234. For code 000235, 000236, or 000237, indexing for the effective address uses register R[5], R[6], or R[7], respectively. Condition Code is set on the loaded quadrupleword. For binary coding details, refer to [Table B-6](#) on page B-7.

**QMPY (000242).** Quadruple Multiply. The quadrupleword integer contained in HGFE is multiplied in quadrupleword integer form by the quadrupleword integer in DCBA. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the result is greater than  $2^{63}-1$  or less than  $-2^{63}$ . Carry can be set, and Condition Code is set on the result.

**QNEG (000244).** Quadruple Negate. The quadrupleword integer contained in DCBA is replaced with its two's complement. Overflow is set if the original operand was  $-2^{63}$ . Condition Code is set on the result.

**QRND (000263).** Quadruple Round. Five is added to the operand in DCBA if the operand is positive ( $-5$  is added if negative), and the result is divided by 11. The new value replaces the former contents of DCBA. Condition Code is set, and the Overflow bit is cleared. For binary coding details, refer to [Table B-6](#) on page B-7.

**QST (00023-).** Quadruple Store. The quadrupleword operand contained in EDCB is stored in the effective memory location indicated by A plus 4 times the index value. No indexing occurs for code 000230. For code 000231, 000232, or 000233, indexing for the effective address uses register R[5], R[6], or R[7], respectively. The quadrupleword operand and A are then deleted from the stack. For binary coding details, refer to [Table B-6](#) on page B-7.

QSUB (000241). Quadruple Subtract. The quadrupleword integer contained in DCBA is subtracted in quadrupleword integer form from the quadrupleword integer in HGFE. Both operands are deleted, and the quadrupleword result is pushed onto the stack. Overflow is set if the result is greater than  $2^{63}-1$  or less than  $-2^{63}$ . Carry is set if no borrow-out occurs. Condition Code is set on the result.

QUP (00025-). Quadruple Scale Up. The operand value in DCBA is multiplied by a specified power of ten (1, 2, 3, or 4), and the new value replaces the former contents of DCBA. Overflow is set if the result is greater than  $2^{63}-1$  or less than  $-2^{63}$ . Condition Code is set on the result. For binary coding details, see [Table B-6](#) on page B-7.

RCLK (000050). Read Clock. This instruction reads the quadrupleword microsecond counter (located in the system data segment), adds the instantaneous value of the 14-bit hardware microsecond counter to it, and pushes the result onto the register stack. Note that because the software counter is updated only every 10 milliseconds (each time the hardware counter rolls over), adding the hardware count to it provides an accurate clock indication at the instant that RCLK is executed. This instruction is nonprivileged.

RCPU (000051). Read CPU Number. This instruction reads this processor's processor number from bits 0:7 of INTB and pushes this value onto the register stack. This is a privileged instruction.

RDE (000024). Read ENV Into A. The contents of the Environment (ENV) register are pushed onto the register stack.

RDP (000025). Read P Into A. The contents of the Program Counter (P) are pushed onto the register stack.

RSUB (025---). Return From Subprocedure. This instruction is used to return from a subprocedure called by a BSUB instruction. The instruction assumes that the return address is on the top of the memory stack (indicated by S) and returns control to that address. S is set to  $S - S^{\wedge}\text{decrement}$ . " $S^{\wedge}\text{decrement}$ " can be any number from 0 to 255. However, to delete the return address from the stack, it must be at least 1. For binary coding details, see [Table B-3](#) on page B-3.

RSW (000026). Read the Switch Register Into A. The contents of the processor's Switch register (always zero on TNS/R processors) are pushed onto the register stack. Condition Code is set. This instruction is nonprivileged.

SBA (000365). Store Byte via A. The byte in B is stored into the effective memory location pointed to by the byte address in A. Both B and A are then deleted. SBA accesses the current data segment only.

SBAR (00017-). Subtract A From a Register. A is subtracted in signed integer form from the register pointed to by the Register field of the instruction. A is deleted from the stack. Overflow is set if the result is greater than 32767 or less than  $-32768$ . Carry can be set, meaning no borrow. Condition Code is set on the result. For binary coding details, see [Table B-5](#) on page B-5.



**SBAS (000355).** Store Byte via A Into System. The byte in B is stored into the effective memory location pointed to by the byte address in A. Both B and A are then deleted. A refers to an address in the system data segment.

**SBRA (00015-).** Subtract Register From A. The contents of the register pointed to by the Register field of the instruction are subtracted in integer form from register A. Overflow is set if the result is greater than 32767 or less than -32768. Carry can be set, meaning no borrow. Condition Code is set on the result. For binary coding details, refer to [Table B-5](#) on page B-5.

**SBU (1266--).** Scan Bytes Until. The SBU instruction expects A.<8:15> to contain a test byte and B to contain the byte address of the string to be scanned. The segment to be used is determined by the “S” field of the instruction and by the DS, CS, and LS bits of the ENV register. If the source address is specified as current code segment, the byte address is taken to be in the same 64K-byte half of the code space as the current P register value. If the source address is specified as latest user code segment, the byte address is taken to be in the lower 64K-byte half of that segment. The “RL” field of the instruction determines whether the scan address will be incremented (“RL” = 0) or decremented (“RL” = 1) after each comparison. The scan is terminated when either a null byte is found in the string or the test byte matches a byte in the string. The Carry (K) bit is set in the ENV register when null byte termination occurs. In either case, B points to the byte address that caused the scan to cease. RP is set to the “RP” field of the instruction at termination. For binary coding details, refer to [Table B-3](#) on page B-3. [Figure 11-7](#) on page 11-9 provides a comparison of ascending and descending directions.

**SBW (1264--).** Scan Bytes While. The SBW instruction expects A to contain a comparison byte in bits 8:15 and B to contain the byte address of the string to be scanned. The segment to be used is determined by the “S” field of the instruction and by the DS, CS, and LS bits of the ENV register. If the source address is specified as latest user code segment, the byte address is taken to be in the lower 64K-byte half of that segment. The “RL” field of the instruction determines whether the source address will be incremented (“RL” = 0) or decremented (“RL” = 1) after each comparison. The scan is terminated when either a null byte is found in the string or a byte in the string does not match the test byte in A. When null byte termination occurs, the Carry (K) bit in the ENV register is set. In either termination case, B points to the byte address that caused termination. RP is set to the “RP” field of the instruction at instruction termination. For binary coding details, refer to [Table B-3](#) on page B-3. [Figure 11-7](#) on page 11-9 provides a comparison of ascending and descending directions.

**SBX (000407).** Store Byte Extended. The byte in bits 8 through 15 of C is stored into the memory location specified by the 32-bit byte address in registers B and A. C, B, and A are then deleted.

**SBXX (0257--, 0267--).** Store Byte Extended, Indexed. The byte contained in B.<8:15> is stored into a computed memory location. The memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value, which is either G[0] (coded 0257--) or the current L register value (coded 0267--); the data word so indicated is assumed to be



the first word of a two-word extended memory pointer. The index value in A is then added to the extended memory pointer to address the location that is to receive the byte being stored. For binary coding details, refer to [Table B-3](#) on page B-3.

SCS (000444). Set Code Segment. Registers B and A are assumed to contain a 17-bit byte address within the current code segment. The doubleword BA is converted from a 17-bit C-relative offset in the current code segment to the corresponding extended relative address. This is done by inserting the appropriate segment number into the upper 15 bits.

SDA (000363). Store Double via A. The doubleword in CB is stored into the effective memory locations starting at the location pointed to by the address in A. C, B, and A are then deleted. SDA accesses the current data segment only.

SDAS (000353). Store Double via A Into System. The doubleword in CB is stored into the effective memory locations starting at the location pointed to by the address in A. C, B, and A are then deleted. A refers to an address in the system data segment.

SDDX (000413). Store Doubleword Extended. The doubleword in registers D and C is stored into memory starting at the location specified by the 32-bit even-byte address in registers B and A. All four words are then deleted from the register stack.

SETE (000022). Set ENV With A. The least significant eight bits of the Environment (ENV) register are replaced with the lower eight bits of the A register. The most significant eight bits of the Environment register are logically ANDed with the upper eight bits of the A register. An instruction-fail interrupt occurs if the new value of ENV has the invalid Condition Code of “negative zero” (N=1, Z=1), or if the DS, CS, or LS bits were altered. This instruction can only clear the PRIV bit of the Environment register, and cannot set it. This instruction is nonprivileged.

SETL (000020). Set L With A. The contents of the L register, which points to the current stack marker, are replaced with the contents of register A. A is then deleted from the register stack.

SETP (000023). Set P With A. The contents of the Program Counter (P) are replaced with the contents of the A Register. A is deleted from the stack, and control is transferred to the new location indicated by P.

SETS (000021). Set S With A. The contents of the S register, which points to the top word of the stack in memory, are replaced with the contents of register A. A is then deleted from the stack. A stack overflow trap occurs if the result is greater than 32767.

SQAS (000446). Store Quadrupleword via A to SG. The quadrupleword in registers EDCB is stored into the four memory locations starting at the location pointed to by the address in A. The address in A refers to an address in the system data segment. All five words are then deleted from the register stack.

SQX (000415). Store Quadrupleword Extended. The quadrupleword in registers FEDC is stored into memory (8 bytes) starting at the location specified by the 32-bit even-byte address in registers B and A. All six words are then deleted from the register stack.

SSW (000027). Store A Into Switch Register. The contents of the A register are set into sysstack[%122]. A is then deleted. This instruction is nonprivileged even though it alters an SG cell.

STAR (00011-). Store A in a Register. The A register contents are stored in the register pointed to by the Register field of the instruction. A is then deleted from the stack. (For binary coding details, refer to [Table B-5](#) on page B-5.)

STB (-54---). Store Byte From A to Data Space. The contents of the byte in bits 8:15 of A are stored in the effective memory location.

STD (-64---). Store Doubleword From BA Into Data Space. The contents of BA are stored in the effective memory location. BA is deleted. [Figure 11-2](#) on page 11-3 illustrates the addressing methods for doubleword instructions. For binary coding details, refer to [Table B-1](#) on page B-1.

STOR (-44---). Store A Into Data Space. The contents of the A register are stored into the effective memory location. A is then deleted from the stack. For binary coding details, refer to [Table B-1](#) on page B-1.

STRP (00010-). Set RP. The register pointer is set to the value in the register field of the instruction. For binary coding details, see [Table B-5](#) on page B-5.

SWA (000361). Store Word via A. The word contained in B is stored into the effective memory location pointed to by the address in A. Both words are then deleted from the stack. SWA accesses the current data segment only.

SWAS (000351). Store Word via A Into System. The word contained in B is stored into the effective memory location pointed to by the address in A. Both words are then deleted from the stack. A refers to an address in the system data segment.

SWX (000411). Store Word Extended. The word in register C is stored into the memory location specified by the 32-bit even-byte address in registers B and A. C, B, and A are then deleted.

SWXX (0255--, 0265--). Store Word Extended, Indexed. The word contained in B is stored into a computed memory location. The memory address is obtained as follows. The displacement value (0 through 63) in bits 10 through 15 of the instruction word is added to a base value, which is either G[0] (coded 0255--) or the current L register value (coded 0265--); the data word so indicated is assumed to be the first word of a two-word even-byte extended memory pointer. The index value in A is sign-extended, then shifted left one bit position (multiplication by 2, because this instruction requires word addressing rather than byte addressing), and then added to the extended memory pointer to address the location that is to receive the word being stored. For binary coding details, refer to [Table B-3](#) on page B-3.

XCAL (127---). External Procedure Call. The XCAL instruction is used to invoke procedures that are outside the current code segment. Like PCAL, XCAL creates a three-word stack marker. Then control is transferred to an instruction in the external segment by a three-step sequence: (1) a number in the XEP field of the instruction refers to an entry in the XEP table of the current code segment, (2) the XEP entry specifies a segment and a PEP entry in that segment, and (3) the PEP entry of the

other code segment specifies a procedure entry point within that segment. The address space mappings of relative segments 2 and 3 (current code segment and latest user code segment) are updated. See detailed descriptions in [Section 6, TNS Execution Modes](#).

XMSK (000064). Exchange MASK With A. The contents of the MASK register are interchanged with the contents of the A register. This is a privileged instruction.

XOR (000012). Logical Exclusive OR A With B. The two words in A and B of the register stack are combined by a logical exclusive OR. The two words are then deleted, and the result is pushed onto the stack. Condition Code is set. (For an example, see [Figure 11-5](#) on page 11-6.)

XSMG (000343). Compute Checksum in Current Data. Starting at the address defined in register B, for a count of words defined in register A, the XSMG instruction exclusive-ORs each word into register C. When the count goes to zero, the two top words on the stack are deleted, leaving the final checksum in register A. The address in B refers to the current data segment only.

XSMX (000333). Compute Checksum Extended. Starting at the memory location defined by the 32-bit even-byte address in CB, for a count of words defined in register A, the XSMX instruction exclusive-ORs each word into register D. When the count goes to zero, the three top words on the stack are deleted, leaving the final checksum in register A.

ZERD (000002). Zero Double. A doubleword zero is pushed onto the top of the register stack (BA). Condition Code is set.

## Additional Operating-System-Only Instructions

The following groups of instructions, most of them privileged, are used solely to implement certain operating system and diagnostic functions in millicode. These instructions are not intended for use in any user applications.

## Resource Management

MXON	(000040)	Mutual Exclusion On
MXFF	(000041)	Mutual Exclusion Off
SFRZ	(000053)	System Freeze
DOFS	(000057)	Disk Record Offset
DLEN	(000070)	Disk Record Length
HALT	(000074)	Processor Halt
PSEM	(000076)	“P” a Semaphore
VSEM	(000077)	“V” a Semaphore
FRST	(000405)	Firmware Reset
RPT	(000442)	Read Process Time
SPT	(000443)	Set Process Timer
BCLD	(000452)	Bus Cold Load
DDTX	(000456)	DDT Request

## Memory Management

CRAX	(000423)	Convert Relative to Absolute Extended Address
------	----------	---

## List Management

DLTE	(000054)	Delete Element From List
INSR	(000055)	Insert Element Into List
MRL	(000075)	Merge Onto Ready List

## Tracing

TRCE	(000217)	Add Entry to Trace Table
------	----------	--------------------------

# A TNS Instruction Lists

This appendix provides two tables that list all instructions in the TNS instruction set with their mnemonics and opcodes, first in alphabetic order by mnemonic and then grouped by type of instruction. The tables are:

[Table A-1, Alphabetic List of Instructions](#), on page A-1

[Table A-2, Categorized List of Instructions](#), on page A-9

For some instructions, the six-digit opcode notation used in [Table A-1](#) and [Table A-2](#) cannot give complete information about the opcode. For instance, the distinctions between QUP and QDWN, ORRI and ORLI, and LWP and LBP cannot be clearly shown. For complete binary coding information, refer to the entries for these instructions in [Appendix B, TNS Instruction Binary Coding](#).

As indicated in the footnotes for these tables, a single asterisk (\*) following an instruction description denotes a privileged instruction and a double asterisk (\*\*) denotes an instruction intended only for operating system use.

---

**Table A-1. Alphabetic List of Instructions** (page 1 of 8)

Mnemonic	Octal Code	Description
ADAR	00016—	Add A to Register
ADDI	104—	Add Immediate
ADDS	002—	Add to S
ADM	—74—	Add to Memory
ADRA	00014—	Add Register to A
ADXl	104—	Add to Index Immediate
ALS	0302—	Arithmetic Left Shift
ANG	000044	AND to Memory
ANLI	007—	AND Left Immediate
ANRI	006—	AND Right Immediate
ANS	000034	AND to SG Memory
ANX	000046	AND to Extended Memory
ARS	0303—	Arithmetic Right Shift
BANZ	—154—	Branch on A Nonzero
BAZ	—144—	Branch on A Zero
BCLD	000452	Bus Cold Load *
BEQL	—12—	Branch if Equal
BFI	000030	Branch Forward Indirect
BGEQ	—13—	Branch if Greater or Equal

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated only for operating system use.

---

**Table A-1. Alphabetic List of Instructions** (page 2 of 8)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
BGTR	—11—	Branch if Greater
BIC	—10—	Branch if Carry
BLEQ	—16—	Branch if Less or Equal
BLSS	—14—	Branch if Less
BNEQ	—15—	Branch if Not Equal
BNOC	—17—	Branch if No Carry
BNOV	—164—	Branch if No Overflow
BOX	—1—4—	Branch on Index
BPT	000451	Instruction Breakpoint Trap
BSUB	—174—	Branch to Subprocedure
BTST	000007	Byte Test
BUN	—104—	Branch Unconditionally
CAQ	000262	Convert ASCII to Quad
CAQV	000261	Convert ASCII to Quad with Initial Value
CCE	000016	Condition Code Equal to
CCG	000017	Condition Code Greater Than
CCL	000015	Condition Code Less Than
CDE	000334	Convert Doubleword to Extended Float
CDF	000306	Convert Doubleword to Float
CDFR	000326	Convert Doubleword to Float (Round)
CDG	000366	Count Duplicate Words
CDI	000307	Convert Doubleword to Integer
CDQ	000265	Convert Doubleword to Quad
CDX	000356	Count Duplicate Words Extended
CED	000314	Extended Float to Doubleword
CEDR	000315	Extended Float to Doubleword (Round)
CEF	000276	Extended Float to Float
CEFR	000277	Extended Float to Float (Round)
CEI	000337	Extended Float to Integer
CEIR	000316	Extended Float to Integer (Round)
CEQ	000322	Extended Float to Quadrupleword
CEQR	000323	Extended Float to Quadrupleword (Round)
CFD	000312	Floating to Doubleword

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated only for operating system use.

**Table A-1. Alphabetic List of Instructions** (page 3 of 8)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
CFDR	000313	Floating to Doubleword (Round)
CFE	000325	Floating to Extended Float
CFI	000311	Floating to Integer
CFIR	000310	Floating to Integer (Round)
CFQ	000320	Floating to Quadrupleword
CFQR	000321	Floating to Quadrupleword (Round)
CID	000327	Convert Integer to Doubleword
CIE	000332	Convert Integer to Extended Float
CIF	000331	Convert Integer to Floating
CIQ	000266	Convert Integer to Quad
CLQ	000267	Convert Logical to Quad
CMBX	000422	Compare Bytes Extended
CMPI	001—	Compare Immediate
COMB	1262—	Compare Bytes
COMW	0262—	Compare Words
CQA	000260	Convert Quad to ASCII
CQD	000247	Convert Quad to Doubleword
CQE	000336	Convert Quad to Extended
CQER	000335	Convert Quad to Extended (Round)
CQF	000324	Convert Quad to Floating
CQFR	000330	Convert Quad to Floating (Round)
CQI	000264	Convert Quad to Integer
CQL	000246	Convert Quad to Logical
CRAX	000423	Convert Relative to Absolute Extended *
DADD	000220	Double Add
DALS	1302—	Double Arithmetic Left Shift
DARS	1303—	Double Arithmetic Right Shift
DCMP	000225	Double Compare
DDIV	000223	Double Divide
DDTX	000456	DDT Request *
DDUP	000006	Double Duplicate
DFG	000367	Deposit Field in Memory
DFS	000357	Deposit Field in SG

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated only for operating system use.

**Table A-1. Alphabetic List of Instructions** (page 4 of 8)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
DFX	000416	Deposit Field in Extended Memory
DISP	000073	Dispatch *
DLEN	000070	Disk Record Length **
DLLS	1300—	Double Logical Left Shift
DLRS	1301—	Double Logical Right Shift
DLTE	000054	Delete Element From List *
DMPY	000222	Double Multiply
DNEG	000224	Double Negate
DOFS	000057	Disk Record Offset **
DPCL	000032	Dynamic Procedure Call
DPF	000014	Deposit Field
DSUB	000221	Double Subtract
DTST	000031	Double Test
DXCH	000005	Double Exchange
EADD	000300	Extended Floating-Point Add
ECMP	000305	Extended Floating-Point Compare
EDIV	000303	Extended Floating-Point Divide
EMPY	000302	Extended Floating-Point Multiply
ENEG	000304	Extended Floating-Point Negate
ESE	000471	Extensible Stack Enter
ESUB	000301	Extended Floating-Point Subtract
EXCH	000004	Exchange
EXIT	125—	Exit Procedure
FADD	000270	Floating-Point Add
FCMP	000275	Floating-Point Compare
FDIV	000273	Floating-Point Divide
FMPY	000272	Floating-Point Multiply
FNEG	000274	Floating-Point Negate
FRST	000405	Firmware Reset *
FSUB	000271	Floating-Point Subtract
HALT	000074	Processor Halt *
IADD	000210	Integer Add
ICMP	000215	Integer Compare

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated only for operating system use.



**Table A-1. Alphabetic List of Instructions** (page 5 of 8)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
IDIV	000213	Integer Divide
IDX1	000344	Calculate Index, 1 Dimension
IDX2	000345	Calculate Index, 2 Dimensions
IDX3	000346	Calculate Index, 3 Dimensions
IDXD	000317	Calculate Index, Bounds in Data Space
IDXP	000347	Calculate Index, Bounds in Code Space
IMPY	000212	Integer Multiply
INEG	000214	Integer Negate
INSR	000055	Insert Element Into List *
ISUB	000211	Integer Subtract
LADD	000200	Logical Add
LADI	003—	Logical Add Immediate
LADR	—7—	Load Address
LAND	000010	Logical AND
LBA	000364	Load Byte via A
LBAS	000354	Load Byte via A From System
LBP	—2—4—	Load Byte From Program
LBX	000406	Load Byte Extended
LBXX	0256—, 0266—	Load Byte Extended, Indexed
LCMP	000205	Logical Compare
LDA	000362	Load Double via A
LDAS	000352	Load Double via A From SG
LDB	—5—	Load Byte
LDD	—6—	Load Double
LDDX	000412	Load Double Extended
LDI	100—	Load Immediate
LDIV	000203	Logical Divide
LDLI	005—	Load Left Immediate
LDRA	00013—	Load Register to A
LDX	—3—	Load X
LDXI	10—	Load X Immediate
LLS	0300—	Logical Left Shift

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated only for operating system use.

**Table A-1. Alphabetic List of Instructions** (page 6 of 8)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
LMPY	000202	Logical Multiply
LNEG	000204	Logical Negate
LOAD	—4—	Load
LOR	000011	Logical OR
LQAS	000445	Load Quadrupleword via A From SG *
LQX	000414	Load Quadrupleword Extended
LRS	0301—	Logical Right Shift
LSUB	000201	Logical Subtract
LWA	000360	Load Word via A
LWAS	000350	Load Word via A From SG
LWP	—2—	Load Word From Program
LWUC	000342	Load Word From User Code segment
LWX	000410	Load Word Extended
LWXX	0254—, 0264—	Load Word Extended, Indexed
MBXR	000420	Move Bytes Extended, Reverse
MBXX	000421	Move Bytes Extended, and Checksum
MNDX	000227	Move Words While Not Duplicate, Extended
MNGG	000226	Move Words While Not Duplicate
MOND	000001	Minus One Double
MOVB	126—	Move Bytes
MOVW	026—	Move Words
MRL	000075	Merge Onto Ready List *
MVBX	000417	Move Bytes Extended
MXFF	000041	Mutual Exclusion Off *
MXON	000040	Mutual Exclusion On *
NOP	000000	No Operation
NOT	000013	Not
NSAR	00012—	Nondestructive Store A in a Register
NSTO	—34—	Nondestructive Store
ONED	000003	One Double
ORG	000045	OR to Memory
ORLI	0044—	OR Left Immediate

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated only for operating system use.

**Table A-1. Alphabetic List of Instructions** (page 7 of 8)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
ORRI	004—	OR Right Immediate
ORS	000035	OR to SG Memory
ORX	000047	OR to Extended Memory
PCAL	027—	Procedure Call
POP	124—	Pop From Stack
PSEM	000076	“P” a Semaphore *
PUSH	024—	Push to Stack
QADD	000240	Quad Add
QCMP	000245	Quad Compare
QDIV	000243	Quad Divide
QDWN	00025—	Quad Scale Down
QLD	00023—	Quad Load
QMPY	000242	Quad Multiply
QNEG	000244	Quad Negate
QRND	000263	Quad Round
QST	00023—	Quad Store
QSUB	000241	Quad Subtract
QUP	00025—	Quad Scale Up
RCLK	000050	Read Clock
RCPU	000051	Read Processor Number
RDE	000024	Read ENV Register
RDP	000025	Read P Register
RPT	000442	Read Process Timer *
RSUB	025—	Return From Subprocedure
RSW	000026	Read Switches
SBA	000365	Store Byte via A
SBAR	00017—	Subtract A From a Register
SBAS	000355	Store Byte via A Into SG
SBRA	00015—	Subtract Register From A
SBU	1266—	Scan Bytes Until
SBW	1264—	Scan Bytes While
SBX	000407	Store Byte Extended

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated only for operating system use.

**Table A-1. Alphabetic List of Instructions** (page 8 of 8)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
SBXX	0257—, 0267—	Store Byte Extended, Indexed
SCS	000444	Set Code Segment
SDA	000363	Store Double via A
SDAS	000353	Store Double via A Into SG
SDDX	000413	Store Double Extended
SETE	000022	Set ENV Register
SETL	000020	Set L Register
SETP	000023	Set P Register
SETS	000021	Set S Register
SFRZ	000053	System Freeze *
SPT	000443	Set Process Timer *
SQAS	000446	Store Quadrupleword via A to SG *
SQX	000415	Store Quadrupleword Extended
SSW	000027	Set Switches
STAR	00011—	Store A in Register
STB	—54—	Store Byte
STD	—64—	Store Double
STOR	—44—	Store
STRP	00010—	Set RP
SWA	000361	Store Word via A
SWAS	000351	Store Word via A Into SG
SWX	000411	Store Word Extended
SWXX	0255—, 0265	Store Word Extended, Indexed
TRCE	000217	Add Entry to Trace Table *
VSEM	000077	“V” a Semaphore *
XCAL	127—	External Call
XMSK	000064	Exchange Mask *
XOR	000012	Exclusive OR
XSMG	000343	Compute Checksum in Current Data
XSMX	000333	Checksum Extended Block
ZERD	000002	Zero Double

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated only for operating system use.

**Table A-2. Categorized List of Instructions** (page 1 of 9)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
<b>16-Bit Arithmetic (Top of Register Stack)</b>		
IADD	000210	Integer Add
LADD	000200	Logical Add
ISUB	000211	Integer Subtract
LSUB	000201	Logical Subtract
IMPY	000212	Integer Multiply
LMPY	000202	Logical Multiply
IDIV	000213	Integer Divide
LDIV	000203	Logical Divide
INEG	000214	Integer Negate
LNEG	000204	Logical Negate
ICMP	000215	Integer Compare
LCMP	000205	Logical Compare
CMPI	001—	Integer Compare Immediate
ADDI	104—	Integer Add Immediate
LADI	003—	Logical Add Immediate
<b>32-Bit Signed Arithmetic</b>		
CDI	000307	Convert Double to Integer
CID	000327	Convert Integer to Double
DADD	000220	Double Add
DSUB	000221	Double Subtract
DMPY	000222	Double Multiply
DDIV	000223	Double Divide
DNEG	000224	Double Negate
DCMP	000225	Double Compare
DTST	000031	Double Test
MOND	000001	(Load) Minus One Double
ZERD	000002	(Load) Zero Double
ONED	000003	(Load) One Double
<b>16-Bit Signed Arithmetic (Register Stack Element)</b>		
ADRA	00014—	Add Register to A
SBRA	00015—	Subtract Register From A
ADAR	00016—	Add A to Register

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated for operating system use only.

**Table A-2. Categorized List of Instructions** (page 2 of 9)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
SBAR	00017—	Subtract A From Register
ADXI	104—	Add to Index Immediate
<b>Decimal Arithmetic Load and Store</b>		
QLD	00023—	Quadruple Load
QST	00023—	Quadruple Store
<b>Decimal Integer Arithmetic</b>		
QADD	000240	Quadruple Add
QSUB	000241	Quadruple Subtract
QMPY	000242	Quadruple Multiply
QDIV	000243	Quadruple Divide
QNEG	000244	Quadruple Negate
QCMP	000245	Quadruple Compare
<b>Decimal Arithmetic Scaling and Rounding</b>		
QUP	00025—	Quadruple Scale Up
QDWN	00025—	Quadruple Scale Down
QRND	000263	Quadruple Round
<b>Decimal Arithmetic Conversions</b>		
CQI	000264	Convert Quad to Integer
CQL	000246	Convert Quad to Logical
CQD	000247	Convert Quad to Double
CQA	000260	Convert Quad to ASCII
CIQ	000266	Convert Integer to Quad
CLQ	000267	Convert Logical to Quad
CDQ	000265	Convert Double to Quad
CAQ	000262	Convert ASCII to Quad
CAQV	000261	Convert ASCII to Quad With Initial Value
<b>Floating Point Arithmetic</b>		
FADD	000270	Floating-Point Add
FSUB	000271	Floating-Point Subtract
FMPY	000272	Floating-Point Multiply
FDIV	000273	Floating-Point Divide
FNEG	000274	Floating-Point Negate
FCMP	000275	Floating-Point Compare

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated for operating system use only.

**Table A-2. Categorized List of Instructions** (page 3 of 9)

Mnemonic	Octal Code	Description
----------	------------	-------------

**Extended Floating-Point Arithmetic**

EADD	000300	Extended Floating-Point Add
ESUB	000301	Extended Floating-Point Subtract
EMPY	000302	Extended Floating-Point Multiply
EDIV	000303	Extended Floating-Point Divide
ENEG	000304	Extended Floating-Point Negate
ECMP	000305	Extended Floating-Point Compare

**Floating-Point Conversions**

CEF	000276	Convert Extended to Floating
CEFR	000277	Convert Extended to Floating, Rounded
CFI	000311	Convert Floating to Integer
CFIR	000310	Convert Floating to Integer, Rounded
CFD	000312	Convert Floating to Double
CFDR	000313	Convert Floating to Double, Rounded
CED	000314	Convert Extended to Double
CEDR	000315	Convert Extended to Double, Rounded
CEI	000337	Convert Extended to Integer
CEIR	000316	Convert Extended to Integer, Rounded
CFQ	000320	Convert Floating to Quad
CFQR	000321	Convert Floating to Quad, Rounded
CEQ	000322	Convert Extended to Quad
CEQR	000323	Convert Extended to Quad, Rounded
CFE	000325	Convert Floating to Extended
CIF	000331	Convert Integer to Floating
CDF	000306	Convert Double to Floating
CDFR	000326	Convert Double to Floating, Rounded
CQF	000324	Convert Quad to Floating
CQFR	000330	Convert Quad to Floating, Rounded
CIE	000332	Convert Integer to Extended
CDE	000334	Convert Double to Extended
CQE	000336	Convert Quad to Extended
CQER	000335	Convert Quad to Extended, Rounded

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated for operating system use only.

**Table A-2. Categorized List of Instructions** (page 4 of 9)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
<b>Array Indexing</b>		
IDX1	000344	Calculate Index, 1 Dimension
IDX2	000345	Calculate Index, 2 Dimensions
IDX3	000346	Calculate Index, 3 Dimensions
IDXP	000347	Calculate Index, Bounds in Code Space
IDXD	000317	Calculate Index, Bounds in Data Space
<b>Register Stack Manipulation</b>		
EXCH	000004	Exchange A With B
DXCH	000005	Double Exchange
DDUP	000006	Double Duplicate
STAR	00011—	Store A in a Register
NSAR	00012—	Nondestructive Store A in a Register
LDRA	00013—	Load A From a Register
LDI	100—	Load Immediate
LDXI	10—	Load Index Immediate
LDLI	005—	Load Left Immediate
<b>Boolean Operations</b>		
LAND	000010	Logical AND
LOR	000011	Logical OR
XOR	000012	Exclusive OR
NOT	000013	NOT
ORRI	004—	OR Right Immediate
ORLI	0044—	OR Left Immediate
ANRI	006—	AND Right Immediate
ANLI	007—	AND Left Immediate
<b>Bit Shift and Deposit</b>		
DPF	000014	Deposit Field
LLS	0300—	Logical Left Shift
DLLS	1300—	Double Logical Left Shift
LRS	0301—	Logical Right Shift
DLRS	1301—	Double Logical Right Shift
ALS	0302—	Arithmetic Left Shift
DALS	1302—	Double Arithmetic Left Shift

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated for operating system use only.



**Table A-2. Categorized List of Instructions** (page 5 of 9)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
ARS	0303—	Arithmetic Right Shift
DARS	1303—	Double Arithmetic Right Shift
<b>Byte Test</b>		
BTST	000007	Byte Test
<b>Memory To/From Register Stack</b>		
LWP	—2—	Load Word From Program
LBP	—2—4—	Load Byte From Program
PUSH	024—	Push Registers to Memory
POP	124—	Pop Memory to Registers
LWXX	0254—, 0264—	Load Word Extended, Indexed
SWXX	0255—, 0265—	Store Word Extended, Indexed
LBXX	0256—, 0266—	Load Byte Extended, Indexed
SBXX	0257—, 0267—	Store Byte Extended, Indexed
LDX	—3—	Load Index
NSTO	—34—	Nondestructive Store
LOAD	—4—	Load Word
STOR	—44—	Store Word
LDB	—5—	Load Byte
STB	—54—	Store Byte
LDD	—6—	Load Double
STD	—64—	Store Double
LADR	—7—	Load Address of Variable
ADM	—74—	Add to Memory
<b>Load and Store via Address on Register Stack</b>		
ANS	000034	AND to SG Memory
ORS	000035	OR to SG Memory
ANG	000044	AND to Current Data
ORG	000045	OR to Current Data
ANX	000046	AND to Extended Memory
ORX	000047	OR to Extended Memory

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated for operating system use only.

**Table A-2. Categorized List of Instructions** (page 6 of 9)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
DFS	000357	Deposit Field Into SG
DFG	000367	Deposit Field Into Current Data
DFX	000416	Deposit Field Extended
LWUC	000342	Load Word From User Code Segment
LWAS	000350	Load Word via A From SG
LWA	000360	Load Word via A
SWAS	000351	Store Word via A Into SG
SWA	000361	Store Word via A
LDAS	000352	Load Double via A From SG
LDA	000362	Load Double via A
SDAS	000353	Store Double via A Into SG
SDA	000363	Store Double via A
LBAS	000354	Load Byte via A From SG
LBA	000364	Load Byte via A
SBAS	000355	Store Byte via A Into SG
SBA	000365	Store Byte via A
LBX	000406	Load Byte Extended
SBX	000407	Store Byte Extended
LWX	000410	Load Word Extended
SWX	000411	Store Word Extended
LDDX	000412	Load Doubleword Extended
SDDX	000413	Store Doubleword Extended
LQX	000414	Load Quadrupleword Extended
SQX	000415	Store Quadrupleword Extended
SCS	000444	Set Code Segment
LQAS	000445	Load Quadrupleword via A From SG *
SQAS	000446	Store Quadrupleword via A to SG *
<b>Branching</b>		
BIC	—10—	Branch if Carry
BUN	—104—	Branch Unconditionally
BOX	—1—4—	Branch on Index
BGTR	—11—	Branch if CC Greater
BEQL	—12—	Branch if CC Equal

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated for operating system use only.

**Table A-2. Categorized List of Instructions** (page 7 of 9)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
BGEQ	–13—	Branch if CC Greater or Equal
BLSS	–14—	Branch if CC Less
BAZ	–144—	Branch if A Zero
BNEQ	–15—	Branch if CC Not Equal
BANZ	–154—	Branch if A Not Zero
BLEQ	–16—	Branch if CC Less or Equal
BN OV	–164—	Branch if No Overflow
BN OC	–17—	Branch if No Carry
BFI	000030	Branch Forward Indirect
<b>Moves, Compares, Scans, and Checksum Computations</b>		
MNGG	000226	Move Words While Not Duplicate
CDG	000366	Count Duplicate Words
MOVW	026—	Move Words
MOVB	126—	Move Bytes
COMW	0262—	Compare Words
COMB	1262—	Compare Bytes
SBW	1264—	Scan Bytes While
SBU	1266—	Scan Bytes Until
MNDX	000227	Move Words While Not Duplicate, Extended
XSMX	000333	Checksum Extended Block
XSMG	000343	Compute Checksum in Current Data
CDX	000356	Count Duplicate Words Extended
MVBX	000417	Move Bytes Extended
MBXR	000420	Move Bytes Extended Reverse
MBXX	000421	Move Bytes Extended, and Checksum
CMBX	000422	Compare Bytes Extended
<b>Program Register Control</b>		
SETL	000020	Set L Register
SETS	000021	Set S Register
SETE	000022	Set ENV Register
SETP	000023	Set P Register
RDE	000024	Read ENV Register
RDP	000025	Read P Register

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated for operating system use only.

**Table A-2. Categorized List of Instructions** (page 8 of 9)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
STRP	00010—	Set Register Pointer
ADDS	002—	Add to S Register
CCL	000015	Set CC Less
CCE	000016	Set CC Equal
CCG	000017	Set CC Greater
<b>Routine Calls/Returns</b>		
PCAL	027—	Procedure Call
XCAL	127—	External Procedure Call
DPCL	000032	Dynamic Procedure Call
EXIT	125—	Exit From Procedure
BSUB	—174—	Branch to Subprocedure
RSUB	025—	Return From Subprocedure
ESE	000471	Extensible Stack Enter
<b>Interrupt System</b>		
XMSK	000064	Exchange MASK Register *
DISP	000073	Dispatch *
<b>Input/Output</b>		
RSW	000026	Read Switch Register
SSW	000027	Set Switch Register
<b>Miscellaneous</b>		
NOP	000000	No Operation
RCLK	000050	Read Clock
RCPU	000051	Read Processor Number
BPT	000451	Instruction Breakpoint Trap
<b>Resource Management</b>		
MXON	000040	Mutual Exclusion On *
MXFF	000041	Mutual Exclusion Off *
SFRZ	000053	System Freeze *
DOFS	000057	Disk Record Offset **
DLEN	000070	Disk Record Length **
HALT	000074	Processor Halt *
PSEM	000076	“P” a Semaphore *
VSEM	000077	“V” a Semaphore *

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated for operating system use only.

**Table A-2. Categorized List of Instructions** (page 9 of 9)

<b>Mnemonic</b>	<b>Octal Code</b>	<b>Description</b>
FRST	000405	Firmware Reset *
RPT	000442	Read Process Timer *
SPT	000443	Set Process Timer *
BCLD	000452	Bus Cold Load *
DDTX	000456	DDT Request *
<b>Memory Management</b>		
CRAX	000423	Convert Relative to Absolute Extended *
<b>List Management</b>		
DLTE	000054	Delete Element From List *
INSR	000055	Insert Element Into List *
MRL	000075	Merge Onto Ready List *
<b>Trace and Breakpoints</b>		
TRCE	000217	Add Entry to Trace Table *

\* Indicates a privileged instruction.

\*\* Indicates an instruction designated for operating system use only.



# B TNS Instruction Binary Coding

This appendix provides a number of reference tables that define the binary coding for most of the TNS instructions, grouped according to the coding patterns of the fields of the instruction words. (For example, all memory reference instructions are listed together.) These tables break down each instruction, bit by bit, into its component parts, indicate the operands, results, and ENV register bit settings, and show relationships between similar instructions. A key at the end of each table explains the symbols used.

The following tables are included in this appendix:

[Table B-1, Binary Coding, Memory Reference Instructions](#)

[Table B-2, Binary Coding, Immediate Instructions](#)

[Table B-3, Binary Coding, Move/Shift/Call/Extended Instructions](#)

[Table B-4, Binary Coding, Branch Instructions](#)

[Table B-5, Binary Coding, Stack Instructions](#)

[Table B-6, Binary Coding, Decimal Arithmetic Instructions](#)

[Table B-7, Binary Coding, Floating-Point Instructions](#)

Table B-1. Binary Coding, Memory Reference Instructions (page 1 of 2)																vk cc	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
I		2		0	X	X	0	+/–		—	P	—				LWP	a
I		2		0	X	X	0	+/–		—	P	—				LBP	b
I		3		0	X	X				G,L,SG,S	—					LDX	a
I		3		1	X	X				G,L,SG,S	—					NSTO	
I		4		0	X	X				G,L,SG,S	—					LOAD	a
I		4		1	X	X				G,L,SG,S	—					STOR	
I		5		0	X	X				G,L,SG,S	—					LDB	b
I		5		1	X	X				G,L,SG,S	—					STB	
I		6		0	X	X				G,L,SG,S	—					LDD	a
I		6		1	X	X				G,L,SG,S	—					STD	
I		7		0	X	X				G,L,SG,S	—					LADR	
I		7		1	X	X				G,L,SG,S	—					ADM vk	a
				P+			0			.			.			[0:177]	
				P–			1			.			.			[0:177]	
				G+			0			.			.			[0:377]	
				L+			1			0			.			[0:177]	
				SG			1			1			0			[0:77]	
				L–			1			1			1			[0:37]	
				S–			1			1			1			[0:37]	

**Table B-1. Binary Coding, Memory Reference Instructions** (page 2 of 2)

I [0 : 1] indicates direct or indirect address.

XX [0 : 3] indicates index register selection.

+/- [0 : 1] implies two's complement notation; the sign is extended through bit 0 at execution.

v = Overflow

k = Carry

cc = Condition Codes:

a  $\left\{ \begin{array}{l} \text{CCL (result} < 0) \text{ or (opr1} < \text{opr2)} \\ \text{CCE (result} = 0) \text{ or (opr1} = \text{opr2)} \\ \text{CCG (result} > 0) \text{ or (opr1} > \text{opr2)} \end{array} \right\}$

Note: opr1 is first item pushed on stack, opr2 is second.

b  $\left\{ \begin{array}{l} \text{CCL (ASCII numeric)} \\ \text{CCE (ASCII alphabetic)} \\ \text{CCG (ASCII special)} \end{array} \right\}$

**Table B-2. Binary Coding, Immediate Instructions**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	vk	cc
1		0			0		+/-	←		Operand			→			LDI	a
1		0		0	X	X	+/-	←		Operand			→			LDXI	a
0		0			1		+/-	←		Operand			→			CMPI	a
0		0			2		+/-	←		Operand			→			ADDS	a
0		0			3		+/-	←		Operand			→			LADI	k a
0		0			4		0	←		Operand			→			ORRI	a
0		0			4		1	←		Operand			→			ORLI	a
1		0			4		+/-	←		Operand			→			ADDI	vk a
1		0		1	X	X	+/-	←		Operand			→			ADXI	vk a
0		0			5		+/-	←		Operand			→			LDLI	a
0		0			6		+/-	←		Operand			→			ANRI	a
0		0			7		+/-	←		Operand			→			ANLI	a

XX [0 : 3] indicates index register selection.

+/- [0 : 1] implies two's complement notation; the sign is extended through bit 0 at execution.

v = Overflow

k = Carry

cc = Condition Codes:

a  $\left\{ \begin{array}{l} \text{CCL (result} < 0) \text{ or (opr1} < \text{opr2)} \\ \text{CCE (result} = 0) \text{ or (opr1} = \text{opr2)} \\ \text{CCG (result} > 0) \text{ or (opr1} > \text{opr2)} \end{array} \right\}$

Note: opr1 is first item pushed on stack, opr2 is second.



**Table B-3. Binary Coding, Move/Shift/Call/Extended Instructions**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	vk	cc
1		2			4			New RP		Last Register			Count-1			PUSH	
1		2			4			New RP		Last Register			Count-1			POP	
0		2			5		0	←		SDEC			→			RSUB	
0		2			5		0	←		SDEC			→			EXIT	
0		2			5,6			4		Displacement			→			LWXX	a
0		2			5,6			5		Displacement			→			SWXX	
0		2			5,6			6		Displacement			→			LBXX	b
1		2			5,6			7		Displacement			→			SBXX	
1		2			6		0	0	RL	S	S	D		RP		MOVW	
0		2			6		0	1	RL	S	S	D		RP		COMW	a
0		2			6		0	0	RL	S	S	D		RP		MOVB	
0		2			6		0	1	RL	S	S	D		RP		COMB	a
1		2			6		1	0	RL	S	S	D		RP		SBW	k
1		2			6		1	1	RL	S	S	D		RP		SBU	k
0		2			7			←		PEP			→			PCAL	
1		2			7			←		XEP			→			XCAL	
0		3			0			0		← Shift			Count	→		LLS	a
1		3			0			0		← Shift			Count	→		DLLS	a
0		3			0			1		← Shift			Count	→		LRS	a
1		3			0			1		← Shift			Count	→		DLRS	a
0		3			0			2		← Shift			Count	→		ALS	a
1		3			0			2		← Shift			Count	→		DALS	a
0		3			0			3		← Shift			Count	→		ARS	a
1		3			0			3		← Shift			Count	→		DARS	a

RL (right-left indicator)

0 Left-to-right (increasing addresses)

1 Right-to-left (decreasing addresses)

SDEC = stack S decrement

SS (source map):

00 Current data (relative segment 0)

01 System data (relative segment 1)

10 Current code (relative segment 2)

11 User code (relative segment 3)

D = destination map (data only)

0 Current data

1 System data

PEP = Procedure entry point table

XEP = External entry point table

vk cc: See Table B-4 footnote

**Table B-4. Binary Coding, Branch Instructions**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	vk	cc
I		1			0		0	+/-	←		P				→		BIC
I		1			0		4	+/-	←		P				→		BUN
I		1		0	X	X	4	+/-	←		P				→		BOX
I		1			1		0	+/-	←		P				→		BGTR
I		1			2		0	+/-	←		P				→		BEQL
I		1			3		0	+/-	←		P				→		BGEQ
I		1			4		0	+/-	←		P				→		BLSS
I		1			4		4	+/-	←		P				→		BAZ
I		1			5		0	+/-	←		P				→		BNEQ
I		1			5		4	+/-	←		P				→		BANZ
I		1			6		0	+/-	←		P				→		BLEQ
I		1			6		4	+/-	←		P				→		BNOV
I		1			7		0	+/-	←		P				→		BNOC
I		1			7		4	+/-	←		P				→		BSUB

I [0 : 1] indicates direct or indirect address.

XX [0 : 3] indicates index register selection.

+/- [0 : 1] implies two's complement notation; the sign is extended through bit 0 at execution.

Note: Because the program counter register holds the address of the next instruction, a branch-self instruction (Branch \*) would be coded BUN P-1.

v = Overflow

k = Carry

cc = Condition Codes:

a { CCL (result < 0) or (opr1 < opr2)  
CCE (result = 0) or (opr1 = opr2)  
CCG (result > 0) or (opr1 > opr2) }

Note: opr1 is first item pushed on stack, opr2 is second.

b { CCL (ASCII numeric)  
CCE (ASCII alphabetic)  
CCG (ASCII special) }

**Table B-5. Binary Coding, Stack Instructions** (page 1 of 2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		0			0		Stack Operand Code								
<div><div></div><div></div></div>															
<7:15>				vk cc				<7:15>				vk cc			
0	0	0	NOP						0	5	3	SFRZ *			
0	0	1	MOND			a			0	5	4	DLTE *			
0	0	2	ZERD			a			0	5	5	INSR *			
0	0	3	ONED			a			0	5	7	DOFS ** c			
0	0	4	EXCH			a			0	6	4	XMSK *			
0	0	5	DXCH			a			0	7	0	DLEN **			
0	0	6	DDUP			a			0	7	3	DISP *			
0	0	7	BTST			b			0	7	4	HALT *			
0	1	0	LAND			a			0	7	5	MRL *			
0	1	1	LOR			a			0	7	6	PSEM *			
0	1	2	XOR			a			0	7	7	VSEM *			
0	1	3	NOT			a			1	0	reg	STRP			
0	1	4	DPF			a			1	1	reg	STAR			
0	1	5	CCL			a			1	2	reg	NSAR			
0	1	6	CCE			a			1	3	reg	LDRA a			
0	1	7	CCG			a			1	4	reg	ADRA vk a			
0	2	0	SETL						1	5	reg	SBRA vk			
0	2	1	SETS						1	6	reg	ADAR vk a			
0	2	2	SETE			!!	!		1	7	reg	SBAR vk a			
0	2	3	SETP						2	0	0	LADD k a			
0	2	4	RDE						2	0	1	LSUB k a			
0	2	5	RDP						2	0	2	LMPY v=0 a			
0	2	6	RSW			a			2	0	3	LDIV v a			
0	2	7	SSW						2	0	4	LNEG k a			
0	3	0	BFI						2	0	5	LCMP a			
0	3	1	DTST			a			2	1	0	IADD vk a			
0	3	2	DPCL						2	1	1	ISUB vk a			
0	3	4	ANS			a			2	1	2	IMPY v a			
0	3	5	ORS			a			2	1	3	IDIV v a			
0	4	0	MXON *						2	1	4	INEG vk a			
0	4	1	MXFF *						2	1	5	ICMP a			
0	4	4	ANG			a			2	1	7	TRCE *			
0	4	5	ORG			a			2	2	0	DADD vk a			
0	4	6	ANX			a			2	2	1	DSUB vk a			
0	4	7	ORX			a			2	2	2	DMPY vk a			
0	5	0	RCLK						2	2	3	DDIV vk a			
0	5	1	RCPU *						2	2	4	DNEG vk a			

**Table B-5. Binary Coding, Stack Instructions** (page 2 of 2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0			0			Stack Operand Code								

<7:15>

vk cc

2 2 5 DCMP a

2 2 6 MNGG !

2 2 7 MNDX !

3 3 3 XSMX

3 4 2 LWUC a

3 4 3 XSMG

3 5 0 LWAS a

3 5 1 SWAS

3 5 2 LDAS a

3 5 3 SDAS

3 5 4 LBAS b

3 5 5 SBAS

3 5 6 CDX

3 5 7 DFS a

3 6 0 LWA a

3 6 1 SWA

3 6 2 LDA a

3 6 3 SDA

3 6 4 LBA b

3 6 5 SBA

3 6 6 CDG

3 6 7 DFG a

<7:15>

vk cc

4 0 5 FRST \*

4 0 6 LBX b

4 0 7 SBX

4 1 0 LWX a

4 1 1 SWX

4 1 2 LDDX a

4 1 3 SDDX

4 1 4 LQX a

4 1 5 SQX

4 1 6 DFX a

4 1 7 MVBX

4 2 0 MBXR

4 2 1 MBXX

4 2 2 CMBX !

4 2 3 CRAX \*

4 4 4 SCS

4 4 5 LQAS \*

4 4 6 SQAS \*

4 5 1 BPT

4 5 2 BCLD \*

4 5 3 TPEF \*

\* indicates a privileged instruction.

\*\* indicates an instruction designated only for operating system use.

v = Overflow

k = Carry

cc = Condition Codes:

a { CCL (result < 0) or (opr1 < opr2)  
CCE (result = 0) or (opr1 = opr2)  
CCG (result > 0) or (opr1 > opr2) }

b { CCL (ASCII numeric)  
CCE (ASCII alphabetic)  
CCG (ASCII special) }

c { CCL (channel error or timeout)  
CCE (no error)  
CCG (unusual condition) }

Note: opr1 is first item pushed on stack, opr2 is second.

! = special vk cc meanings; see instruction definitions in Table C-1.

**Table B-6. Binary Coding, Decimal Arithmetic Instructions**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

<7:15>				vk cc		<7:15>				vk cc	
2	3	0	QST			2	5	0	QUP	v	a
2	3	1	QST x5			2	5	1	QDWN	v=0	a
2	3	2	QST x6			2	5	2	QUP (2)	v	a
2	3	3	QST x7			2	5	3	QDWN (2)	v=0	a
2	3	4	QLD		a	2	5	4	QUP (3)	v	a
2	3	5	QLD x5		a	2	5	5	QDWN (3)	v=0	a
2	3	6	QLD x6		a	2	5	6	QUP (4)	v	a
2	3	7	QLD x7		a	2	5	7	QDWN (4)	v=0	a
2	4	0	QADD	vk	a	2	6	0	CQA	v	a
2	4	1	QSUB	vk	a	2	6	1	CAQV	v	!
2	4	2	QMPY	v	a	2	6	2	CAQ	v	!
2	4	3	QDIV	v	a	2	6	3	QRND	v=0	a
2	4	4	QNEG	vk	a	2	6	4	CQI	v	
2	4	5	QCMP		a	2	6	5	CDQ		
2	4	6	CQL	v		2	6	6	CIQ		
2	4	7	CQD	v		2	6	7	CLQ		

x5 = Indexing for the effective address uses register R[5].

x6 = Indexing for the effective address uses register R[6].

x7 = Indexing for the effective address uses register R[7].

(2) = Power of 10 for scaling.

(3) = Power of 10 for scaling.

(4) = Power of 10 for scaling.

v = Overflow

k = Carry

cc = Condition Codes:

a { CCL (result < 0) or (opr1 < opr2)  
CCE (result = 0) or (opr1 = opr2)  
CCG (result > 0) or (opr1 > opr2) }

Note: opr1 is first item pushed on stack, opr2 is second.

! = CCE if entire string is ASCII digits; CCG if not

**Table B-7. Binary Coding, Floating-Point Instructions**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0				0										

<7:15>						<7:15>					
vk cc						vk cc					
2	7	0	FADD	v	a	3	1	6	CEIR		a
2	7	1	FSUB	v	a	3	1	7	IDXD		a
2	7	2	FMPY	v	a	3	2	0	CFQ		a
2	7	3	FDIV	v	a	3	2	1	CFQR		a
2	7	4	FNEG		a	3	2	2	CEQ		a
2	7	5	FCMP		a	3	2	3	CEQR		a
2	7	6	CEF		a	3	2	4	CQF		a
2	7	7	CEFR		a	3	2	5	CFE		a
3	0	0	EADD	v	a	3	2	6	CDFR		a
3	0	1	ESUB	v	a	3	2	7	CID		a
3	0	2	EMPY	v	a	3	3	0	CQFR		a
3	0	3	EDIV	v	a	3	3	1	CIF		a
3	0	4	ENEG		a	3	3	2	CIE		a
3	0	5	ECMP		a	3	3	4	CDE		a
3	0	6	CDF		a	3	3	5	CQER		a
3	0	7	CDI		a	3	3	6	CQE		a
3	1	0	CFIR		a	3	3	7	CEI		a
3	1	1	CFI		a	3	4	4	IDX1		a
3	1	2	CFD		a	3	4	5	IDX2		a
3	1	3	CFDR		a	3	4	6	IDX3		a
3	1	4	CED		a	3	4	7	IDXP		a
3	1	5	CEDR		a						

v = Overflow

k = Carry

cc = Condition Codes:

a { CCL (result < 0) or (opr1 < opr2)  
 CCE (result = 0) or (opr1 = opr2)  
 CCG (result > 0) or (opr1 > opr2) }

Note: opr1 is first item pushed on stack, opr2 is second.

# C TNS Instruction Set Definition

## Symbol Definitions

This appendix defines the TNS instruction set for the NonStop S-series processors, using conventions of the TAL programming language but based upon the special symbols that are defined in [Table C-1](#). Following this table of symbol definitions, [Table C-2](#) on page C-11 provides the formal definitions of the instructions in numeric order according to the octal codes.

**Table C-1. Definitions of Symbols** (page 1 of 9)

Notation	Description
$x \& y =$	bitwise "and" of x and y
$x   y =$	bitwise "or" of x and y
$x \text{ xor } y =$	bitwise "exclusive or" of x and y
$x \text{ mod } y =$	x modulo y
$\sim x =$	bitwise "complement" of x
$x \ll n =$	x arithmetically shifted left n bits; whether sign bit is retained is undefined
$x \gg n =$	x arithmetically shifted right n bits
$x' \ll n =$	x logically shifted left n bits
$x' \gg n =$	x logically shifted right n bits
$x \text{ rotate } n =$	$x' \ll n + x.<0:n-1>$
$x : y =$	if $x < y$ then -1 else if $x = y$ then 0 else 1
$x' < y =$	comparison of x and y as 16-bit unsigned numbers
$x' : y =$	if $x' < y$ then -1 else if $x = y$ then 0 else 1
$x \text{ max } y =$	if $x > y$ then x else y
$x := y =$	exchange x and y
$x^y =$	concatenate x and y. x and y are 16 bits long unless otherwise indicated
$x^y$	the value x is y bits wide
The following functions perform type conversion between integers, double integers, and addresses:	
$\$DBL( x^y ) =$	$( x^0(32-y) \gg (32-y) )^{32}$ signed conversion to an INT32
$\$DBLL(x, y) =$	$(x^y)^{32}$ logical conversion to an INT32
$\$UDBL(x^y) =$	$( 0(32-y)^{x^y} )^{32}$ unsigned conversion to an INT32
$\$XADR( x^y ) =$	$( 0(32-y-1)^{x^y} 0^1 )^{32}$ unsigned conversion of short word address to a 32-bit byte address

**Table C-1. Definitions of Symbols** (page 2 of 9)

Notation	Description
<b>A</b>	
A =	R[RP]
address =	if indirect then \$XADR(xmem[ dir.adr ]) else dir.adr
<b>B</b>	
B =	R[RP-1]
BA =	B.<0:15>^A.<0:15>
BPADDR =	sysstack[ %115:%116 ]
BPADDRX =	sysstack[ %137 ]
BPBASE =	sysstack[ %123 ]
BPLIM =	sysstack[ %125 ]
BPSIZE =	sysstack[ %124 ]
branch =	TNSP:=branch address
branch address=	if indirect then code[dba] + dba else dba
BRT =	sysstack[ %1400:%1777 ]
byteaddress =	if indirect then \$UDBL(xmem[ dir.adr ]+X) else dir.adr + \$DBL(X)
bytedest( la )=	mem[destseg,la]
byteflag =	<8*la.<31>:8*la.<31>+7>
bytesource( la )=	mem[srcseg, curhalfcseg+la]
bytex =	mem[dseg, byteaddress]
bxmem[ x ] =	xmem[x] for 1 byte
<b>C</b>	
C =	R[RP-2]
CB.<0:31> =	C.<0:15>^B.<0:15>
CC.<0:31> Z N	simulated condition codes zero condition negative condition
cc(x)=	Z:=(x=0); N:=(x<0)
ccb(x)=	Z:=( "A"<=x<="Z" ) or ( "a"<=x<="z" ); N:=( "0"<=x<="9" )
ccl(x)=	cc(x); K:=adder carry out; on subtracts, K:= no borrow out
ccn(x)=	ccl(x); V:=adder overflow
CCSEG	CCSEG[0:].<0:31> = base address of latest code segment for UC, SC, UL, and SL spaces
ccz(x)=	Z:=(x=0); N:=0;



**Table C-1. Definitions of Symbols** (page 3 of 9)

Notation	Description
chkp(x)=	if memory location "x" is absent then Page Fault
CLOCK =	sysstack[ %350:%353 ]
code[ la ] =	mem[ CCSEG[ csegx ], \$XADR(la) ]
codeb[ ba ]	mem[ CCSEG[csegx], ba ] for 1 byte
COLDTIME =	sysstack[ %354:%357 ]
computeshiftcount (max) =	if I.<10:15>=0 then {shiftcount:= A.<11:15>; RP:=RP-1} else shiftcount:= \$min(max,I.<10:15>) result undefined if shiftcount <0 or shiftcount > max
CPCB =	sysstack[ 3 ]
CPDS	current process data space
CS =	ENV.<7>
csegx =	LS ^ CS
CSPACECS =	CSPACEID.<7>
CSPACEID.<0:15>=	code space identifier
CSPACEINDEX =	CSPACEID.<11:15>
CSPACELS =	CSPACEID.<4>
curhalfcseg =	if I.<10:11> = 2 ! ss field of MOVb,COMB,SBW,or SBU op ! selects current code segment then P.<0>^(16 zeroes) else 0
<b>D</b>	
D =	R[RP-3]
dba=	P+I.<9:15>-128*I.<8>
DC=	(D^C) <sup>32</sup>
DCBA =	(D^C^B^A) <sup>64</sup>
dest( la ) =	mem[ destseg, \$XADR(la) ]
destseg =	if I.<12> then SG else 0

**Table C-1. Definitions of Symbols** (page 4 of 9)

Notation	Description
dir.adr =	if I.<7> = 0 then 'global variable' \$XADR(I.<8:15>) (0:255) else if I.<8> = 0 then 'local variable' Lx.<0:31>+\$XADR(I.<9:15>) (0:127) else if I.<9> = 0 then 'system global' SG.<0:31>+\$XADR(I.<10:15>) (0:63) else if I.<10> = 0 then 'procedure param' Lx.<0:31>-\$XADR(I.<11:15>) (0:31) else Sx.<0:31>-\$XADR(I.<11:15>); 'subroutine param' (0:31)
DS =	ENV.<6>
dseg =	if I.<7:9> = 6 then SG else 0
dwordx =	mem[dseg,address+\$XADR(X <sup>01</sup> ): address+\$XADR(X <sup>01</sup> )+3 ]]
<b>E</b>	
E =	R[RP-4]
ED =	(E <sup>D</sup> ) <sup>32</sup>
ENV.<0:15> =	Simulated environment register
exp=	temp data word (exponent value)
exponent(x)=	x.<7:15>
<b>F</b>	
F =	R[RP-5]
FE =	(F <sup>E</sup> ) <sup>32</sup>
<b>G</b>	
G =	R[RP-6]
<b>H</b>	
H =	R[RP-7]
HGFE =	(H <sup>G</sup> F <sup>E</sup> ) <sup>64</sup>
<b>I</b>	
I.<0:15> =	Simulated TNS instruction opcode
IFAIL( parm )	TNS Instruction Failure Interrupt 'interrupt via SIV[3]'; 'parm' is the trap code which becomes the parameter in the SIV.
imm=	I.<8:15>-256*I.<7>
indirect =	I.<0>
Indivisible Off	'turn back on external interrupts'.

**Table C-1. Definitions of Symbols** (page 5 of 9)

Notation	Description
Indivisible On	'turn off all external interrupts'. Exceptions for page fault, overflow, IFAIL will still be processed.
INQ[0:1,0:15].<0:15>=	interprocessor bus in queues
INTA.<0:15> =	interrupt register A
INTB.<0:15> =	interrupt register B
IOC =	sysstack[ %2000:%3777 ]
IOSPAD =	IOC scratchpad registers (IOC cache)
IOSPADXLAT =	translate and move the IOC entries
IPDS	interrupt environment's process data space
<b>K</b>	
K	simulated carry flag
<b>L</b>	
Lx.<0:31>	local stack pointer, location of the current TNS stack marker.
LS	ENV.<4>
<b>M</b>	
MASK.<0:15> =	interrupt mask register
mem[sas,la]=	xmem[ sas+la ]
movestep=	if I.<9> then -1 else 1
MYEXTCPU=	sysstack[ %154 ]: <8:11> - Cluster number <12:15>- Processor number
<b>N</b>	
N =	simulated Negative condition code, and/or ENV.<11>
norm(x)=	while x.<0> <> 0 do {x:=x'<<'1; exp:=exp-1}
<b>O</b>	
OUTQ[0:15].<0:15>=	interprocessor bus out queue
overflow=	V:=1; if "divide by zero" then Z:=1 else if exp.<0>=1 then N:=1 else Z:=N:=0
<b>P</b>	
P.<0:15>=	Simulated TNS program counter, word location of current instruction + 1
Px.<0:31>=	Simulated TNS program counter, byte location of current instruction + 2
PDS	process data space
PHYSEG =	physical page to (abs segment, or lock count) table

**Table C-1. Definitions of Symbols** (page 6 of 9)

Notation	Description
PHYTNSSEGSZ =	Size of a physical segment in logical TNS pages. Physical segments directly mapped and do not have a SEG entry.
PRIV =	ENV.<5>, simulated privileged bit
priv trap =	instruction failure interrupt via SIV[3]
PTIME =	sysstack[ 126:127 ]
<b>R</b>	
REGSAVE=	<pre>{   REGS[0:31].&lt;0:31&gt;   ARITHHI.&lt;0:31&gt;;   ARITHLO.&lt;0:31&gt;;   RETPC.&lt;0:31&gt;   FLTPC.&lt;0:31&gt;   CDS.&lt;0:31&gt;   TNSREGSVALID.&lt;0:7&gt;   CSPACEID.&lt;0:15&gt;   MASK.&lt;0:15&gt;   S.&lt;0:15&gt;      Not saved   P.&lt;0:15&gt;      Not saved   ENV.&lt;0:15&gt;    PRIV and DS in TNS ENV format   L.&lt;0:15&gt;      Not saved   TNSREGS[0:7].&lt;0:15&gt; };</pre>
ReturnPc =	The next inline PC address logically after the TNS micro routine.
RLIST =	sysstack[ %100:%101 ]
RP =	ENV.<13:15>
RPHYPTE(frame, tags,ppte) =	<pre>{   frame := ppte.pfn;   tags.&lt;10&gt; := ppte.N;   tags.&lt;11&gt; := ppte.G;   tags.&lt;12&gt; := ppte.P;   tags.&lt;13&gt; := ppte.R;   tags.&lt;14&gt; := ppte.D;   tags.&lt;15&gt; := ~ppte.V; };</pre>
<b>S</b>	
Sx.<0:31> =	stack pointer=location of last word of stack
savearea =	SPAD[xx:xx+yy]
SCSEGBASE =	%h7c00,0000
SG =	if PRIV then %h8002000 else IFAIL

**Table C-1. Definitions of Symbols** (page 7 of 9)

Notation	Description
SIV =	32-bit system interrupt vector <pre>{   Lx.&lt;0:31&gt; GIH stack pointer, INT32 aligned   MASK.&lt;0:15&gt;   FILLER.&lt;0:15&gt;   RETPC.&lt;0:31&gt; PC of the interrupt handler   PMAP.&lt;0:31&gt; address of SC.0's PMAP table   PARM1.&lt;0:31&gt;   PARM2.&lt;0:31&gt;   FILLER.&lt;0:63&gt; };</pre>
source(la) =	mem[ srcseg, \$XADR(la) ]
SPAD =	PDS[%hFFFF9000]
srcsegx =	case I.<10:11> of begin 0;                   ! data SG                 ! system data CCSEG[csegx];       ! current code CCSEG[0];           ! user code end;                 ! all protection is by ! address faults
stack[ la ] =	xmem[ \$XADR( la) ]
stackb[ lba ] =	xmem[ lba ]
sysstack[la] =	xmem[ SG, \$XADR(la) ]
sysstackb[lba] =	mem[ SG, lba ]
<b>T</b>	
T =	ENV.<8>
TNSP.<0:15> =	Simulated TNS program counter, location of current instruction + 1
Trap_Address =	%00
Trap_No_Segment =	%20
Trap_Read_Only =	%21
Trap_Misaligned =	%22
Trap_bad_sysbus_addr =	%23
Trap_Instruction =	%01
Trap_Undefined_TNS_Op=	%40
Trap_Unimplemented_TNS_Op =	%41
Trap_Priv_Op =	%42
Trap_Invalid_Operand =	%43
Trap_Bad_Case =	%44
Trap_Dead_End =	%45
Trap_Undef_Native_Op =	%47
Trap_Arithoverflow =	%02

**Table C-1. Definitions of Symbols** (page 8 of 9)

Notation	Description
Trap_Int_zero_divide =	%60
Trap_Int_Overflow =	%61
Trap_Bounds =	%62
Trap_soft_Overflow =	%63
Trap_Fp_Zero_divide =	%64
Trap_Fp_Exp_Overflow =	%65
Trap_Fp_Exp_Underflow=	%66
TRACE =	sysstack[ %121 ]
TRBASE =	sysstack[ %117 ]
TRLIM =	sysstack[ %120 ]
<b>U</b>	
UC=	ENV.<0>
UCSEGBASE =	%h7000,0000
ULSEGBASE =	%h7200,0000
<b>V</b>	
V	simulated overflow flag
VPTE(ppte)=	vpte := 0 <sup>9</sup> ^ppte.pfn^ppte.r^ppte.d^(~ppte.v)
<b>W</b>	
word =	mem[ dseg, address ]
wordx =	mem[ dseg, address + \$XADR(X) ]
WwhyPTE(ppte,frame, tags)=	{ ppte.pfn := frame ; ppte.N := tag.<10>; ppte.G := tag.<11>; ppte.P := tag.<12>; ppte.R := tag.<13>; ppte.D := tag.<14>; ppte.V := ~tag.<15>; };
<b>X</b>	
X =	if I.<5:6>=0 then 0 else R[I.<5:6>+4]
XB.<0:31> =	simulated extended base address register
xbase=	stack[ Lx*I.<5>+I.<10:15> : Lx*I.<5>+I.<10:15>+1 ]
XL.<0:31> =	simulated extended limit address register
xmem[ x ] =	if DS then IPDS[ x] <sup>16</sup> else CPDS[ x] <sup>16</sup>  IPDS = the special process data space used by all machine interrupt handlers CPDS = the process data space as currently mapped for the current user process

**Table C-1. Definitions of Symbols** (page 9 of 9)

Notation	Description
<code>xmem2[ x ] =</code>	<code>xmem[ x : x+3 ]<sup>32</sup></code>
<code>xmem4[ x ] =</code>	<code>xmem[ x : x+7 ]<sup>64</sup></code>
<code>xmap(x)=</code>	<pre> ! cross code space map: ! x= new spaceid !   x.&lt;4&gt;=      new LS value !   x.&lt;7&gt;=      new CS value !   x.&lt;11:15&gt;=  new segment index { CSPACEID := x; m := CSPACELS*2 + CSPACECS; case m of { ! 0: User code       seg := ucsegbase + CSPACEINDEX*%H2,0000;  ! 1: System code       seg := scsegbase + CSPACEINDEX*%H2,0000;  ! 2: User library       seg := ulsegbase + CSPACEINDEX*%H2,0000;  ! 3: System library       seg := slsegbase + CSPACEINDEX*%H2,0000; }; CCSEG[csegr ] := seg; }; </pre>

# Instruction Definitions

[Table C-2](#) presents symbolic definitions of the TNS instruction set for the NonStop S-series processor, in numeric opcode order. Refer to [Table C-1](#) for definitions of the symbols used in this table. The one-character symbols immediately to the right of the instruction opcodes have the following meanings:

\* indicates privileged instruction.

@ indicates operating system use only.

C indicates that the instruction becomes temporarily privileged.

Also:

op(x) indicates that an operation similar to that performed by the instruction 'op' should be done using the value(s) 'x'.

Compatibility notes (such as “[See Note 1: CALLABLE PRIV]”) are referred to in some of the definitions and are defined at the end of this table.

## Optionally Indivisible On

Indicates that this instruction is atomic (use Indivisible on) in certain implementations when required to do so. Otherwise it is normally implemented as nonatomic. Using the instruction atomically is externally controlled by Accelerator translation options or the special atomic modal instruction sequence.

## Indivisible On

Indicates that this instruction sequence must complete without external interruptions. Some exceptions are allowed. Page fault interrupts are allowed for some operations that use a special RESTART POINT. It is an error for most TNS indivisible instructions to encounter a page fault.

## RESTART POINT

Restart points are a special implementation detail to allow mutex on to work properly. It is a cooperation between trap handlers and the instruction to restart differently than normal. Restart points are used by all optional atomic instructions to restart from the beginning if there are any page faults during the operation.



**Table C-2. Instruction Definitions** (page 1 of 38)

0 0 0 0 0 0	NOP	no operation
0 0 0 0 0 1	MOND	minus one double RP:=RP+2; cc(B:=A:=-1)
0 0 0 0 0 2	ZERD	zero double RP:=RP+2; cc(B:=A:=0)
0 0 0 0 0 3	ONED	one double RP:=RP+2; B:=0; cc(A:=1)
0 0 0 0 0 4	EXCH	exchange A:=:B; cc(A)
0 0 0 0 0 5	DXCH	double exchange BA:=:DC; cc(BA)
0 0 0 0 0 6	DDUP	double duplicate RP:=RP+2; cc(BA:=DC)
0 0 0 0 0 7	BTST	byte test ccb(A.<8:15>); RP:=RP-1
0 0 0 0 1 0	LAND	logical and cc(B:=B&A); RP:=RP-1
0 0 0 0 1 1	LOR	logical or cc(B:=B   A); RP:=RP-1
0 0 0 0 1 2	XOR	exclusive or cc(B:=B xor A); RP:=RP-1
0 0 0 0 1 3	NOT	not cc(A:= ~ A)
0 0 0 0 1 4	DPF	deposit field cc(C:=(C&B   A&~B)); RP:=RP-2
0 0 0 0 1 5	CCL	condition code less Z:=0; N:=1
0 0 0 0 1 6	CCE	condition code equal Z:=1; N:=0
0 0 0 0 1 7	CCG	condition code greater Z:=N:=0
0 0 0 0 2 0	SETL	set L register Lx:= \$XADR(A); RP:=RP-1
0 0 0 0 2 1	SETS	set S register Sx:= \$XADR(A); if Sx '>=' %200000 then 'stack overflow'; RP:=RP-1

**Table C-2. Instruction Definitions (page 2 of 38)**

0 0 0 0 2 2	SETE	set ENV register ENV.RP := A.<13:15>; ENV.<0:7> := ENV.<0:7> & A.<0:7>; if ENV.DS <> A.DS then 'IFAIL(TRAP_INVALID_OPERAND)'; if (ENV.<0:7> & ~A.<0:7>) <> 0 then IFAIL(TRAP_INVALID_OPERAND); 'validate ENV.NZ'; case A.<11:12> of {cc(1);           ! N,Z = (0,0) cc(0);           ! N,Z = (0,1) cc(-1);          ! N,Z = (1,0) 'IFAIL';         ! N,Z = (1,1); }; ENV.V := A.<10>; K := A.<9>; ENV.T := A.<8>; if ENV.T & ENV.V then 'overflow'; if ENV.RP <> predicted RP then 'choose Nonacclerated Mode'
0 0 0 0 2 3	SETP	set P register TNSP := A ; RP:=RP-1 if 'P is register-exact point' & RP = 7 then 'choose Acclerated Mode' else 'choose Nonaccelerated Mode';
0 0 0 0 2 4	RDE	read ENV register RP:=RP+1; A := old ENV [ Note : ENV holds a local copy of PRIV and DS ]
0 0 0 0 2 5	RDP	read P register RP:=RP+1; A:=P;
0 0 0 0 2 6	RSW	read switches RP:=RP+1; cc(A:=0)
0 0 0 0 2 7 C	SSW	set switches sysstack[%122]:=A; RP:=RP-1 [ See Note 1: CALLABLE PRIV ;]
0 0 0 0 3 0	BFI	branch forward indirect TNSP:=TNSP+A+code[TNSP+A]; RP:=RP-1
0 0 0 0 3 1	DTST	double test cc(BA)

**Table C-2. Instruction Definitions (page 3 of 38)**

0 0 0 0 3 2	DPCL	dynamic procedure call
<pre> A.&lt;0:6&gt; - space id A.&lt;7:15&gt; - pep index  t := (ENV &amp; %177740)   CSPACEID ; stack[S+1:S+3] := (Px.&lt;15:30&gt;,t,Lx.&lt;15:30&gt;); t.&lt;7&gt; := A.&lt;0&gt;;           ! CS t.&lt;4&gt; := A.&lt;1&gt;;           ! LS t.&lt;11:15&gt; := A.&lt;2:6&gt;;      ! space index xmap( t ); m := A.&lt;0&gt; + 2*A.&lt;1&gt; + 2; t := A.&lt;7:15&gt;; p := t.&lt;7:15&gt;; if ~ PRIV then     {if p&gt;= mem[CCSEG[m],0] then         {'validate CCSEG with a trusted copy';          if p &gt;= mem[CCSEG[m],2]              then priv trap;           PRIV:=1         }     }; Lx:=Sx:=Sx+3*2; if Lx &gt;= %200000 then 'stack overflow'; CS:=t.&lt;0&gt;; LS:=t.&lt;1&gt;; P:=code[t.&lt;7:15&gt;]; RP:=7 if 'P is register-exact point' then     'choose Accelerated Mode' else     'choose Nonaccelerated Mode'; </pre>		
0 0 0 0 3 4	ANS	and to SG memory
<pre> 'Optionally Indivisible  On'; cc(sysstack[A] := sysstack[A] &amp; B); RP:=RP-2 'Optionally Indivisible  Off';  [ See Note 2: PRIV ONLY ] </pre>		
0 0 0 0 3 5	ORS	or to SG memory
<pre> 'Optionally Indivisible  On'; cc(sysstack[A]:=sysstack[A]   B); RP:=RP-2 'Optionally Indivisible  Off'  [ See Note 2: PRIV ONLY ] </pre>		

**Table C-2. Instruction Definitions** (page 4 of 38)

0 0 0 0 4 0*	MXON	mutex on A=<0:7> INTNS code size <8:15> INTNS stack size 'Restart Point:' 'Indivisible On'; if chkp(stackb[Lx-\$XADR(20)) max 0]) then {page fault; goto 'restart point'}; if chkp(stackb[Sx+\$XADR(A.<8:15>)]) then {page fault; goto 'restart point'}; if A.<0:7> then if chkp(codeb[TNSP+\$XADR(A.<0:7>)]) then {page fault; goto 'restart point'}; stackb[Lx+\$XADR(1)]:=MASK; MASK:=MASK & %177640; RP:=RP-1 'Indivisible Off'
0 0 0 0 4 1*	MXFF	mutex off MASK:=stackb[Lx+\$XADR(1)]
0 0 0 0 4 4	ANG	and to memory 'Optionally Indivisible On'; c(stack[A]:=stack[A] & B); P:=RP-2 'Optionally Indivisible Off'  [ See Note 1: CALLABLE PRIV ]
0 0 0 0 4 5	ORG	or to memory 'Optionally Indivisible On'; c(stack[A]:=stack[A]   B); P:=RP-2 'Optionally Indivisible Off'  [ See Note 1: CALLABLE PRIV ]
0 0 0 0 4 6	ANX	and to extended memory 'Optionally Indivisible On'; c(xmem[BA]:=xmem[BA] & C); P:=RP-3 'Optionally Indivisible Off'  [ See Note 1: CALLABLE PRIV ]
0 0 0 0 4 7	ORX	or to extended memory 'Optionally Indivisible On'; cc(xmem[BA]:=xmem[BA]   C); RP:=RP-3 'Optionally Indivisible Off'  [ See Note 1: CALLABLE PRIV ]
0 0 0 0 5 0	RCLK	read clock 'Indivisible On'; RP:=RP+4; DCBA:= CLOCK + COLDTIME + TIMER; 'Indivisible Off'  CLOCK = sysstack[%350:%353] COLDTIME = sysstack[%354:%357 ]  [ See Note 1: CALLABLE PRIV ] [ See Note 3: CLOCK ADDR ]

**Table C-2. Instruction Definitions (page 5 of 38)**

0 0 0 0 5 1	RCPU	<p>read processor #  RP:=RP+1; A:=processor #</p> <p>[ See Note 2: PRIV ONLY ]</p>
0 0 0 0 5 3*	SFRZ	<p>system freeze  assert system freeze; halt</p>
0 0 0 0 5 4*	DLTE	<p>deletes an element from a doubly  linked, circular list</p> <p>A=element address</p> <p>'Indivisible On';  if sysstack[A] &lt;&gt; 0 then  {if sysstack[sysstack[A]+1]  &lt;&gt; A or  sysstack[sysstack[A+1]]  &lt;&gt; A  then IFAIL( _Invalid_Operand );  f:=sysstack[A];  b:=sysstack[A+1];  sysstack[b]:=f;  sysstack[f+1]:=b;  sysstack[A]:=0;  sysstack[A+1]:=0;  };  RP:=RP-1  'Indivisible Off'  !!! Note !!!</p> <p>all memory locations accessed  must be present</p>
0 0 0 0 5 5*	INSR	<p>inserts an element into a doubly  linked, circular list</p> <p>B=list header  A=list element</p> <p>'Indivisible On';  if A=0 or  sysstack[sysstack[B]+1]  &lt;&gt; B or  sysstack[sysstack[B+1]]  &lt;&gt; B  then IFAIL( _Invalid_Operand );  f:=sysstack[B];  sysstack[B]:=A;  sysstack[A]:=f;  sysstack[A+1]:=B;  sysstack[f+1]:=A;  RP:=RP-2  'Indivisible Off'  !!! Note !!!</p> <p>all memory locations accessed  must be present</p>

**Table C-2. Instruction Definitions** (page 6 of 38)

0 0 0 0 5 7@	DOFS	<p>disk record offset A=record number; on completion, A holds offset into buffer of record</p> <p>if A &gt;= '512 or (A:=xmem[stackb[Lx+4:Lx+6]-A*2]) ' &gt;= 'stackb[Lx+8] then {TNSP:=stackb[Lx+10]; RP:=7};</p>
0 0 0 0 6 4*	XMSK	<p>exchange mask 'Indivisible On'; MASK:=A 'Indivisible Off'</p>
0 0 0 0 7 0@	DLEN	<p>disc record length A=record number</p> <p>if (A:=DOFS(A+1)-DOFS(A)) &lt; 0 then {TNSP:=stackb[Lx+10]; RP:=7}</p>
0 0 0 0 7 3*	DISP	<p>dispatch 'Indivisible On'; set dispatcher interrupt; siv[15].parml_lo.&lt;15&gt;:=1 'Indivisible Off'</p>
0 0 0 0 7 4*	HALT	<p>halt halt</p>
0 0 0 0 7 5*	MRL	<p>merge onto ready list BA: xaddr of PCB element to merge</p> <p>'Indivisible On'; new_pri := bxmem[BA+9]; prev := xaddr of sysstack[%132:%133]; do { next := prev; prev := xmem2[prev+4]; } until bxmem[prev+9] &gt;= new_pri; if xmem2[prev] &lt;&gt; next then IFAIL; xmem2[prev] := BA; xmem2[next+4] := BA; xmem2[BA] := next; xmem2[BA+4] := prev; cur_pri := if CPCB = 0xFFFFC0000 then 0 else bxmem[CPCB+9]; if cur_pri &lt; new_pri then DISPATCH; 'Indivisible Off' RP:=RP-2</p>

**Table C-2. Instruction Definitions** (page 7 of 38)

0	0	0	0	7	6*	PSEM	"P" a semaphore DC=wait time BA=semaphore extended absolute address  'Indivisible On'; xmem[BA+4]:=xmem[BA+4]-1; if xmem[BA+4] < 0 then {\$R[4] := DC; \$R[5] := BA; siv[15].parml_lo:=siv[15].parml_lo   5; set dispatcher interrupt; 'Indivisible Off'; D := \$R[3].<16:31>;     ! \$V0 } else {D:=1; xmem[BA+6]:=CPCB 'Indivisible Off'; }; RP:=RP-3
0	0	0	0	7	7*	VSEM	"V" a semaphore BA=semaphore extended absolute address  'Indivisible On'; xmem[BA+4]:=xmem[BA+4]+1; if xmem[BA+4] <= 0 then {set dispatcher interrupt; siv[15].parml_lo.<12>:=1 \$R[4] := BA; } else xmem[BA+6]:=0; 'Indivisible Off'; RP:=RP-2
0	0	0	1	0	reg	STRP	set RP RP:=reg
0	0	0	1	1	reg	STAR	store A in reg R[reg]:=A; RP:=RP-1
0	0	0	1	2	reg	NSAR	nondestructive store A in reg R[reg]:=A
0	0	0	1	3	reg	LDRA	load register to A RP:=RP+1; cc(A:=R[reg])
0	0	0	1	4	reg	ADRA	add register to A ccn(A:=A+R[reg])
0	0	0	1	5	reg	SBRA	subtract register from A ccn(A:=A-R[reg])
0	0	0	1	6	reg	ADAR	add A to register ccn(R[reg]:=R[reg]+A); RP:=RP-1
0	0	0	1	7	reg	SBAR	subtract A from register ccn(R[reg]:=R[reg]-A); RP:=RP-1
0	0	0	2	0	0	LADD	logical add ccl(B:=B+A); RP:=RP-1
0	0	0	2	0	1	LSUB	logical subtract ccl(B:=B-A); RP:=RP-1

**Table C-2. Instruction Definitions (page 8 of 38)**

0 0 0 2 0 2	LMPY	logical multiply cc(BA:=B*'A); V:=0
0 0 0 2 0 3	LDIV	logical divide V:=(C>='A); (C,B):=(CB 'mod' A,CB '/' A); cc(B); RP:=RP-1  [ See Note 4: OVERFLOW RESULTS UNDEFINED ]  A is undefined after the operation
0 0 0 2 0 4	LNEG	logical negate ccl(A:=-A)
0 0 0 2 0 5	LCMP	logical compare cc(B:'A); RP:=RP-2
0 0 0 2 1 0	IADD	integer add ccn(B:=B+A); RP:=RP-1  [ See Note 5: OVFL TRAP RESULTS UNDEFINED ] [ See Note 6: OVERFLOW TNS COMPATIBLE ]
0 0 0 2 1 1	ISUB	integer subtract ccn(B:=B-A); RP:=RP-1  [ See Note 5: OVFL TRAP RESULTS UNDEFINED ] [ See Note 6: OVERFLOW TNS COMPATIBLE ]
0 0 0 2 1 2	IMPY	integer multiply V:=~(-32768<=B*A<=32767); cc(B:=B*A); RP:=RP-1  [ See Note 3: OVERFLOW RESULTS UNDEFINED ]
0 0 0 2 1 3	IDIV	integer divide V:=~(-32768<=B/A<=32767); cc(B:=B/A); RP:=RP-1  [ See Note 3: OVERFLOW RESULTS UNDEFINED ]
0 0 0 2 1 4	INEG	integer negate ccn(A:=-A)
0 0 0 2 1 5	ICMP	integer compare cc(B:A); RP:=RP-2
0 0 0 2 1 7*	TRCE	add an entry to the trace table HGFEDCBA=entry  'Indivisible On'; if TRBASE<'TRLIM then {sysstack[TRACE:TRACE+7] := HGFEDCBA; TRACE:=TRACE+8; if TRACE>'TRLIM then TRACE:=TRBASE}; 'Indivisible Off'
0 0 0 2 2 0	DADD	double add ccn(DC:=DC+BA); RP:=RP-2.  [ See Note 5: OVFL TRAP RESULTS UNDEFINED ] [ See Note 6: OVERFLOW TNS COMPATIBLE ]



**Table C-2. Instruction Definitions (page 9 of 38)**

0 0 0 2 2 1	DSUB	double subtract ccn(DC:=DC-BA); RP:=RP-2  [ See Note 5: OVFL TRAP RESULTS UNDEFINED ] [ See Note 6: OVERFLOW TNS COMPATIBLE ]
0 0 0 2 2 2	DMPY	double multiply ccn(DC:=DC*BA); K:=0; RP:=RP-2  [ See Note 4: OVERFLOW RESULTS UNDEFINED]
0 0 0 2 2 3	DDIV	double divide cc(DC:=DC/BA); K:=0; V:=(BA=0 or DC=2**31); RP:=RP-2  [ See Note 4: OVERFLOW RESULTS UNDEFINED]
0 0 0 2 2 4	DNEG	double negate ccn(BA:=-BA)
0 0 0 2 2 5	DCMP	double compare cc(DC:BA); RP:=RP-4
0 0 0 2 2 6	MNGG	move words while not duplicate D=destination C=source B=count A=value <> to value of source  while cc(B)<>"=" and stack[C]<>A do {A:=stack[D]:=stack[C]; D:=D+1; C:=C+1; B:=B-1; }; RP:=RP-1
0 0 0 2 2 7	MNDX	move words while not duplicate extended FE=destination DC=source B=count A=value<>to value of source  while cc(B)<>"=" and xmem[DC]<>A do {A:=xmem[FE]:=xmem[DC]; FE:=FE+2; DC:=DC+2; B:=B-1; }; RP:=RP-1
0 0 0 2 3 0xx	QST	quad store adr:=(if I=%230 then 0 else R[I.<14:15>+4])*4+A; stack[adr:adr+3]:=EDCB; RP:=RP-5  [ See Note 7: NOT ATOMIC ]
0 0 0 2 3 4xx	QLD	quad load adr:=(if I=%234 then 0 else R[I.<14:15>+4])*4+A; RP:=RP+3; cc(DCBA:=stack[adr:adr+3])

**Table C-2. Instruction Definitions** (page 10 of 38)

0 0 0 2 4 0	QADD	quad add ccn(HGFE:=HGFE + DCBA); RP:=RP-4  [ See Note 5: OVFL TRAP RESULTS UNDEFINED ] [ See Note 6: OVERFLOW TNS COMPATIBLE ]
0 0 0 2 4 1	QSUB	quad subtract ccn(HGFE:=HGFE - DCBA); RP:=RP-4  [See Note 5: OVFL TRAP RESULTS UNDEFINED ] [See Note 6: OVERFLOW TNS COMPATIBLE ]
0 0 0 2 4 2	QMPY	quad multiply V:= -2**63<=HGFE*DCBA<=2**63-1; HGFE:=HGFE * DCBA; cc(HGFE); RP:=RP-4  [ See Note 4: OVERFLOW RESULTS UNDEFINED]
0 0 0 2 4 3	QDIV	quad divide V:=if DCBA=0 or (HGFE = -2**63 & DCBA = -1) then 1 else 0; HGFE:=HGFE / DCBA; cc(HGFE); RP:=RP-4  [ See Note 4: OVERFLOW RESULTS UNDEFINED]
0 0 0 2 4 4	QNEG	quad negate DCBA:=-DCBA; ccn(DCBA)
0 0 0 2 4 5	QCMP	quad compare cc(HGFE:DCBA)
0 0 0 2 4 6	CQL	convert quad to logical V:=if 0 <= DCBA <=2**16-1 then 0 else 1; D:=A; RP:=RP-3
0 0 0 2 4 7	CQD	convert quad to double V:=if -2**31 <=DCBA<= 2**31-1 then 0 else 1; DC:=BA; RP:=RP-2
0 0 0 2 5 nn0	QUP	quad scale up DCBA:=DBCA* 10**(I.<13:14>+1); V:=if -2**63<=DCBA<=2**63-1 then 0 else 1; cc(DCBA) [ See Note 4: OVERFLOW RESULTS UNDEFINED]
0 0 0 2 5 nn1	QDWN	quad scale down DCBA:=DBCA/ 10**(I.<13:14>+1); V:=0; cc(DCBA);

**Table C-2. Instruction Definitions (page 11 of 38)**

0 0 0 2 6 0	CQA	convert quad to ASCII cc(FEDC); FEDC := abs(FEDC); B:=B+A; while A<>0 do {B:=B-1; bytedest(B):= %60+abs(FEDC) mod 10; FEDC:=FEDC/10; A:=A-1 }; V:=if FEDC=0 then 0 else 1; RP:=RP-6
0 0 0 2 6 1	CAQV	convert ASCII to quad with initial value DCBA - initial value  V:=0; N:=1; while E<>0 and V=0 and N=1 do {ccb(t:=bytedest(F)); if N=1 then {DCBA:=DCBA*10 + t&%17; V:=if DCBA<=2**63-1 then 0 else 1; F:=F+1; E:=E-1 } }; cc(E) !cce if entire string !is ASCII digits. !ccg if not. Note: initial value (DCBA) should be positive.
0 0 0 2 6 2	CAQ	convert ASCII to quad RP:=RP+4; DCBA:=0; V:=0; N:=1; while E<>0 and V=0 and N=1 do {ccb(t:=bytedest(F)); if N=1 then {DCBA:=DCBA*10 + t&%17; V:=if DCBA<=2**63-1 then 0 else 1; F:=F+1; E:=E-1 } }; cc(E) !cce if entire string is ASCII digits. !ccg if not.
0 0 0 2 6 3	QRND	quad round DCBA:=(if DCBA<0 then DCBA-5 else DCBA+5) / 10; V:=0; cc(DCBA)
0 0 0 2 6 4	CQI	convert quad to integer V:=if -2**15 <=DCBA<= 2**15-1 then 0 else 1; D:=A; RP:=RP-3;

**Table C-2. Instruction Definitions** (page 12 of 38)

0 0 0 2 6 5	CDQ	convert double to quad (t,u):=BA; s:=if B<0 then %177777 else 0; RP:=RP+2; DCBA:=(s,s,t,u)
0 0 0 2 6 6	CIQ	convert integer to quad t:=A; s:=if A<0 then %177777 else 0; RP:=RP+3; DCBA:=(s,s,s,t)
0 0 0 2 6 7	CLQ	convert logical to quad t:=A;RP:=RP+3; DCBA:=(0,0,0,t)
0 0 0 2 7 0	FADD	floating add V:=0 t1:=exponent(C); t2:=exponent(A); if BA<>0 and DC<>0 and abs(t1-t2)<=24 then {sign1:=D.<0>; sign2:=B.<0>; D.<0>:=B.<0>:=1; exponent(C):=0; exponent(A):=0; s:=t1-t2; if s>=0 then BA:=BA'>>'s; else {DC:=DC'>>'-'s; DC:=BA; t1:=t2} if sign1=sign2 then {DC:=DC+'BA; if carry then {DC:=DC'>>'1; t1:=t1+1; D.<0>:=1}} else {DC:=DC-'BA; if not carry then {DC:=-DC; sign1:=~sign1} if DC=0 then t1:=sign1:=0 else while D.<0>=0 do {DC:=DC'<<'1; t1:=t1-1}} DC:=DC+'%400; if carry then t1:=t1+1; if t1.<6>=1 then call overflow; D.<0>:=sign1; exponent(C):=t1} else if DC=0 or t1-t2<-24 then DC:=BA; if no overflow then cc(DC); RP:=RP-2

**Table C-2. Instruction Definitions** (page 13 of 38)

0 0 0 2 7 1	FSUB	floating subtract if BA<>0 then B.<0>:=~B.<0>; goto FADD
0 0 0 2 7 2	FMPY	floating multiply V:=0 if DC=0 or BA=0 then DC:=0 else {t1:=exponent(C); t2:=exponent(A); exp:=t1+t2-255; sign:=D.<0> xor B.<0>; D.<0>:=B.<0>:=1; exponent(C):=0; exponent(A):=0; DCBA:=DC*'BA; norm(DC); DC:=DC+'%400; if carry out then exp:=exp+1; if exp.<6>=1 then call overflow; D.<0>:=sign; exponent(C):=exp} if no overflow then cc(DC); RP:=RP-2
0 0 0 2 7 3	FDIV	floating divide V:=0 if BA=0 then call overflow; else if DC<>0 then {t1:=exponent(C); t2:=exponent(A); exp:=t1-t2+256; sign:=D.<0> xor B.<0>; D.<0>:=B.<0>:=1; exponent(C):=0; exponent(A):=0; DC:=DC/'BA; norm(DC); DC:=DC+'%400; if carry out then exp:=exp+1; if exp.<6>=1 then call overflow; D.<0>:=sign; exponent(C):=exp} if no overflow then cc(DC); RP:=RP-2
0 0 0 2 7 4	FNEG	floating negate V := 0; if BA<>0 then B.<0>:=~B.<0>; cc(BA)

**Table C-2. Instruction Definitions** (page 14 of 38)

0 0 0 2 7 5	FCMP	floating compare V := 0; if D.<0> <> B.<0> then cc(D:B) else {sign:=D.<0>; D.<0>:=B.<0>:=0; t1:=exponent(C); t2:=exponent(A); if t1<>t2 then if sign=0 then cc(t1:t2) else cc(t2:t1) else if sign=0 then cc(DC:BA) else cc(BA:DC)} RP:=RP-4
0 0 0 2 7 6	CEF	convert extended to floating V := 0; exponent(C):=exponent(A); RP:=RP-2
0 0 0 2 7 7	CEFR	convert extended to floating with rounding V := 0; sign:=D.<0>; D.<0>:=1; exp:=exponent(A); DC:=DC+'%400; if carry then {exp:=exp+1; if exp.<6> then V:=1} D.<0>:=sign; exponent(C):=exp; RP:=RP-2

**Table C-2. Instruction Definitions** (page 15 of 38)

0 0 0 3 0 0	EADD	extended add V:=0 t1:=exponent(E); t2:=exponent(A); if DCBA<>0 and HGFE<>0 and abs(t1-t2)<56 then {sign1:=H.<0>; sign2:=D.<0>; H.<0>:=D.<0>:=1; exponent(E):=0; exponent(A):=0; s:=t1-t2; if s>=0 then DCBA:=DCBA'>>'s; else {HGFE:=HGFE'>>'-'s; HGFE:=DCBA; t1:=t2} if sign1=sign2 then {HGFE:=HGFE+'DCBA; if carry then {HGFE:=HGFE'>>'1; t1:=t1+1; H.<0>:=1}} else {HGFE:=HGFE-'DCBA; if not carry then {HGFE:=-HGFE; sign1:=~sign1} if HGFE=0 then t1:=sign1:=0 else while H.<0>=0 do {HGFE:=HGFE'<<'1; t1:=t1-1}} HGFE:=HGFE+'%400; if carry out then t1:=t1+1; if t1.<6>=1 then call overflow; H.<0>:=sign1; exponent(E):=t1} else if HGFE=0 or t1-t2<=-56 then HGFE:=DCBA; if no overflow then cc(HGFE); RP:=RP-4
0 0 0 3 0 1	ESUB	extended subtract if DCBA<>0 then D.<0>:=~D.<0>; goto EADD

**Table C-2. Instruction Definitions** (page 16 of 38)

0 0 0 3 0 2	EMPY	extended multiply V:=0 if HGFE=0 or DCBA=0 then HGFE:=0 else {t1:=exponent(E); t2:=exponent(A); exp:=t1+t2-255; sign:=H.<0> xor D.<0>; H.<0>:=D.<0>:=1; exponent(E):=0; exponent(A):=0; HGFE:=HGFE*'DCBA'; norm(HGFE); HGFE:=HGFE+'%400'; if carry out then exp:=exp+1; if exp.<6>=1 then call overflow; H.<0>:=sign; exponent(E):=exp} if no overflow then cc(HGFE); RP:=RP-4  [ See Note 15: EMPY Greater Precision ]
0 0 0 3 0 3	EDIV	extended divide V:=0 if DCBA=0 then call overflow; else if HGFE<>0 then {t1:=exponent(E); t2:=exponent(A); exp:=t1-t2+256; sign:=H.<0> xor D.<0>; H.<0>:=D.<0>:=1; exponent(E):=0; exponent(A):=0; HGFE:=HGFE/'DCBA'; norm(HGFE); HGFE:=HGFE+'%400'; if carry out then exp:=exp+1; if exp.<6>=1 then call overflow; H.<0>:=sign; exponent(E):=exp} if no overflow then cc(HGFE); RP:=RP-4
0 0 0 3 0 4	ENEG	extended negate V := 0; if DCBA<>0 then D.<0>:=~D.<0>; cc(DCBA)



**Table C-2. Instruction Definitions** (page 17 of 38)

0 0 0 3 0 5	ECMP	extended compare V := 0; if H.<0> <> D.<0> then cc(H:D) else {sign:=H.<0>; H.<0>:=D.<0>:=0; t1:=exponent(E); t2:=exponent(A); if t1<>t2 then if sign=0 then cc(t1:t2) else cc(t2:t1) else if sign=0 then cc(HGFE:DCBA) else cc(DCBA:HGFE)}
0 0 0 3 0 6	CDF	convert double to floating V := 0; sign:=B.<0>; exp:=31+256; if sign=1 then BA:=-BA; if BA<>0 then {norm(BA); exponent(A):=exp; B.<0>:=sign}
0 0 0 3 0 7	CDI	convert double to integer V := 0; if B+A.<0> <> 0 then V:=1 else V:=0; B:=A; RP:=RP-1
0 0 0 3 1 0	CFIR	convert floating to integer with rounding V := 0; t:=15+256-exponent(A); sign:=B.<0>; if -2**15 <= BA <= 2**15-1 then {B.<0>:=1; BA:=BA'>>'t; BA:=BA+'%100000; if sign=1 then B:=-B else if B.<0>=1 then V:=1} else V:=1; cc(B); RP:=RP-1
0 0 0 3 1 1	CFI	convert floating to integer V := 0; t:=15+256-exponent(A); sign:=B.<0>; if -2**15 <= BA <= 2**15-1 then {B.<0>:=1; BA:=BA'>>'t; if sign=1 then B:=-B} else V:=1; cc(B); RP:=RP-1

**Table C-2. Instruction Definitions (page 18 of 38)**

0 0 0 3 1 2	CFD	convert floating to double $V := 0;$ $t := 31 + 256 - \text{exponent}(A);$ $\text{sign} := B.<0>;$ if $-2^{**31} \leq BA \leq 2^{**31}-1$ then $\{B.<0>:=1;$ $\text{exponent}(A):=0;$ $BA:=BA'>>'t;$ if $\text{sign}=1$ then $BA:=-BA\}$ else $V:=1;$ cc(BA)
0 0 0 3 1 3	CFDR	convert floating to double with rounding $V := 0;$ $t := 31 + 256 - \text{exponent}(A);$ $\text{sign} := B.<0>;$ if $-2^{**31} \leq BA \leq 2^{**31}-1$ then $\{B.<0>:=1;$ $\text{exponent}(A):=0;$ $BAs:=BAs'>>'t;$ $BAs:=BAs+' \%1000000;$ if $\text{sign}=1$ then $BA:=-BA$ else if $B.<0>=1$ then $V:=1\}$ else $V:=1;$ cc(BA)
0 0 0 3 1 4	CED	convert extended to double $V := 0;$ $t := 31 + 256 - \text{exponent}(A);$ $\text{sign} := D.<0>;$ if $-2^{**31} \leq DCBA \leq 2^{**31}-1$ then $\{D.<0>:=1;$ $DC:=DC'>>'t;$ if $\text{sign}=1$ then $DC:=-DC\}$ else $V:=1;$ cc(DC); RP:=RP-2
0 0 0 3 1 5	CEDR	convert extended to double with rounding $V := 0;$ $t := 31 + 256 - \text{exponent}(A);$ $\text{sign} := D.<0>;$ if $-2^{**31} \leq DCBA \leq 2^{**31}-1$ then $\{D.<0>:=1;$ $DCB:=(DCB'>>'t) '+' \%1000000;$ if $\text{sign}=1$ then $DC:=-DC$ else if $D.<0>=1$ then $V:=1\}$ else $V:=1;$ cc(DC); RP:=RP-2

**Table C-2. Instruction Definitions** (page 19 of 38)

0 0 0 3 1 6	CEIR	convert extended to integer with rounding <pre> V := 0; t:=15+256-exponent(A); sign:=D.&lt;0&gt;; if -2**15 &lt;= DCBA &lt;= 2**15-1   then {D.&lt;0&gt;:=1;         DC:=(DC'&gt;&gt;'t) '+' %100000;         if sign=1 then D:=-D         else if D.&lt;0&gt;=1 then V:=1}       else V:=1; cc(D); RP:=RP-3 </pre>
0 0 0 3 1 7	IDXD	calculate index offset and test for bounds violation (bounds table in data space) <pre> V := 0; t:=stack[A]; bc:=t.&lt;0&gt;; t.&lt;0&gt;:=0; indv:=0; psize:=1; s:=A; if t &lt;&gt; predicted number of dimensions then   use Nonaccelerated Mode while t&gt;0 do   {lower:=stack[s:=s+1];   upper:=stack[s:=s+1];   if B&lt;lower and bc=0 then     {V:=1; t =0;     cc(-1); R[7]:=B}   if B&gt;upper and bc=0 then     {V:=1; t =0;     cc(1); R[7]:=B}   size:=upper-lower+1;   B:=B-lower;   indv:=indv+psize*B;   psize:=psize*size;   RP:=RP-1; t:=t-1} if V=0 then   {R[7]:=indv;   cc(R[7])}; RP:=RP-1 </pre>
0 0 0 3 2 0	CFQ	convert floating to quad <pre> V := 0; t:=63+256-exponent(A); sign:=B.&lt;0&gt;; RP:=RP+2; if -2**63 &lt;= DC &lt;= 2**63-1   then {D.&lt;0&gt;:=1;         exponent(C):=0;         B:=A:=0;         DCBA:=DCBA'&gt;&gt;'t;         if sign=1 then DCBA:=-DCBA}       else V:=1; cc(DCBA) </pre>

**Table C-2. Instruction Definitions** (page 20 of 38)

0 0 0 3 2 1	CFQR	convert floating to quad with rounding V := 0; t:=63+256-exponent(A); sign:=B.<0>; RP:=RP+2; if -2**63 <= DC <= 2**63-1 then {D.<0>:=1; exponent(C):=0; B:=A:=s:=0; DCBAs:=(DCBAs'>>'t) '+' %1000000; if sign=1 then DCBA:=-DCBA} else V:=1; cc(DCBA)
0 0 0 3 2 2	CEQ	convert extended to quad V := 0; t:=63+256-exponent(A); sign:=D.<0>; if -2**63 <= DCBA <= 2**63-1 then {D.<0>:=1; exponent(A):=0; DCBA:=DCBA'>>'t; if sign=1 then DCBA:=-DCBA} else V:=1; cc(DCBA)
0 0 0 3 2 3	CEQR	convert extended to quad with rounding V := 0; t:=63+256-exponent(A); sign:=D.<0>; if -2**63 <= DCBA <= 2**63-1 then {D.<0>:=1; exponent(A):=0; s:=0; DCBAs:=(DCBAs'>>'t) '+' %1000000; if sign=1 then DCBA:=-DCBA} else V:=1; cc(DCBA)
0 0 0 3 2 4	CQF	convert quad to floating V := 0; sign:=D.<0>; exp:=63+256; if sign=1 then DCBA:=-DCBA; if DCBA<>0 then {norm(DCBA); exponent(C):=exp; D.<0>:=sign} RP:=RP-2
0 0 0 3 2 5	CFE	convert floating to extended V := 0; G:=exponent(A); exponent(A):=0; H:=0; RP:=RP+2

**Table C-2. Instruction Definitions** (page 21 of 38)

0 0 0 3 2 6	CDFR	convert double to floating with rounding V := 0; sign:=B.<0>; exp:=31+256; if sign=1 then BA:=-BA; if BA<>0 then {norm(BA); BA:=BA+'%400; if carry out then exp:=exp+1; exponent(A):=exp; B.<0>:=sign}
0 0 0 3 2 7	CID	convert integer to double H:=A; A:=A>>15; V:=0; RP:=RP+1
0 0 0 3 3 0	CQFR	convert quad to floating with rounding V := 0; sign:=D.<0>; exp:=63+256; if sign=1 then DCBA:=-DCBA; if DCBA<>0 then {norm(DCBA); DC:=DC+'%400; if carry then exp:= exp+1; exponent(C):=exp; D.<0>:=sign} RP:=RP-2
0 0 0 3 3 1	CIF	convert integer to floating V := 0; sign:=A.<0>; exp:=15+256; if sign=1 then A:=-A; if A<>0 then {norm(A); H:=exp; A.<0>:=sign} else H:=0; RP:=RP+1
0 0 0 3 3 2	CIE	convert integer to extended V := 0; sign:=A.<0>; exp:=15+256; if sign=1 then A:=-A; H:=G:=0; if A<>0 then {norm(A); F:=exp; A.<0>:=sign} else F:=0; RP:=RP+3
0 0 0 3 3 3	XSMX	checksum extended block D=initial checksum CB=block address A=count  while A<>0 do {D:=D xor xmem[CB]; A:=A-1; CB:=CB+2}; RP:=RP-3

**Table C-2. Instruction Definitions** (page 22 of 38)

0 0 0 3 3 4	CDE	convert double to extended V := 0; sign:=B.<0>; exp:=31+256; if sign=1 then BA:=-BA; H:=0; if BA<>0 then {norm(BA); G:=exp; B.<0>:=sign} else G:=0; RP:=RP+2
0 0 0 3 3 5	CQER	convert quad to extended with rounding V := 0; sign:=D.<0>; exp:=63+256; if sign=1 then DCBA:=-DCBA; if DCBA<>0 then {norm(DCBA); DCBA:=DCBA+'%400; if carry out then exp:=exp+1; exponent(A):=exp; D.<0>:=sign}
0 0 0 3 3 6	CQE	convert quad to extended V := 0; sign:=D.<0>; exp:=63+256; if sign=1 then DCBA:=-DCBA; if DCBA<>0 then {norm(DCBA); exponent(A):=exp; D.<0>:=sign}
0 0 0 3 3 7	CEI	convert extended to integer V := 0; t:=15+256-exponent(A); sign:=D.<0>; if -2**15 <= DCBA <= 2**15-1 then {D.<0>:=1; D:=D'>>'t; if sign=1 then D:=-D} else V:=1; cc(D); RP:=RP-3
0 0 0 3 4 2	LWUC	load word from user code space cc(A:=mem[CCSEG[0],\$XADR(A)])
0 0 0 3 4 3	XSMG	checksum block C=initial checksum B=block address A=count  while A<>0 do {C:=C xor stack[B]; A:=A-1; B:=B+1}; RP:=RP-2

**Table C-2. Instruction Definitions** (page 23 of 38)

0 0 0 3 4 4	IDX1	<p>calculate index offset and test index bounds for 1 dimension  (bounds table in code space)  V := 0;  lower:=code[A];  upper:=code[A+1];  if B&lt;lower then  V:=1;  if B&gt;upper then  {V:=1;  R[7]:=B-lower;  cc(R[7])}  RP:=RP-2</p> <p>[ See Note 18: Range error ]</p>
0 0 0 3 4 5	IDX2	<p>calculate index offset and test index bounds for 2 dimensions  (bounds table in code space)  V :=0;  lower:=code[A];  upper:=code[A+1];  if B&lt;lower then  V:=1;  if B&gt;upper then  {V:=1;  s:=upper-lower+1;  B:=B-lower;  lower:=code[A+2];  upper:=code[A+3];  if C&lt;lower then  {V:=1;  if C&gt;upper then  {V:=1;  R[7]:=(C-lower)*s+B;  cc(R[7])}  RP:=RP-3</p> <p>[ See Note 18: Range error ]</p>
0 0 0 3 4 6	IDX3	<p>calculate index offset and test index bounds for 3 dimensions  (bounds table in code space)  V := 0;  indv:=0; psize:=1;  for i=1 to 3 by 1 do  {lower:=code[A];  upper:=code[A:=A+1];  if B&lt;lower then  {V:=1;  if B&gt;upper then  {V:=1;  size:=upper-lower+1;  B:=B-lower;  indv:=indv+psize*B;  psize:=psize*size;  B:=A+1;  RP:=RP-1}  {R[7]:=indv;  cc(R[7])}  RP:=RP-1</p> <p>[ See Note 18: Range error ]</p>

**Table C-2. Instruction Definitions** (page 24 of 38)

0 0 0 3 4 7	IDXP	<p>calculate index offset and test indices for bounds violation (bounds table in code space)</p> <pre> V := 0; t:=code[A]; bc:=t.&lt;0&gt;; t.&lt;0&gt;:=0; indv:=0; psize:=1; s:=A; if t &lt;&gt; predicted number of dimensions then use Nonaccelerated Mode while t&gt;0 do {lower:=code[s:=s+1]; upper:=code[s:=s+1]; if B&lt;lower and bc=0 then {V:=1; if B&gt;upper and bc=0 then {V:=1; size:=upper-lower+1; B:=B-lower; indv:=indv+psize*B; psize:=psize*size; RP:=RP-1; t:=t-1} {R[7]:=indv; cc(R[7])} RP:=RP-1 </pre> <p>[ See Note 18: Range error ]</p>
0 0 0 3 5 0	LWAS	<p>load SG int16 via A cc(A:=sysstack(A))</p> <p>[ See Note 2: PRIV ONLY ]</p>
0 0 0 3 5 1	SWAS	<p>stor SG word via A sysstack[A]:=B; RP:=RP-2</p> <p>[ See Note 2: PRIV ONLY ]</p>
0 0 0 3 5 2	LDAS	<p>load SG double via A RP:=RP+1; cc(BA:=sysstack[B:B+1]) [ See Note 7: NOT ATOMIC ] [ See Note 2: PRIV ONLY ]</p>
0 0 0 3 5 3	SDAS	<p>store SG double via A sysstack[A:A+1]:=CB; RP:=RP-3;</p> <p>[ See Note 2: PRIV ONLY ]</p>
0 0 0 3 5 4	LBAS	<p>load SG byte via A ccb(A:=bytesource(A))</p> <p>[ See Note 2: PRIV ONLY ]</p>
0 0 0 3 5 5	SBAS	<p>store SG byte via A bytedest(A):=B; RP:=RP-2</p> <p>[ See Note 2: PRIV ONLY ]</p>



**Table C-2. Instruction Definitions** (page 25 of 38)

0 0 0 3 5 6	CDX	count duplicate words extended DC=buffer address B=buffer size A=duplicate count  while B<>0 and xmem[DC]=xmem[DC-2] do {A:=A+1; B:=B-1; DC:=DC+2 }
0 0 0 3 5 7	DFS	deposit field in SG memory 'Optionally Indivisible On'; cc(sysstack[A]:= (sysstack[A] & ~B)   (C & B)); RP:=RP-3 'Indivisible Off'  [ See Note 2: PRIV ONLY ]
0 0 0 3 6 0	LWA	load word via A cc(A:=stack[A])
0 0 0 3 6 1	SWA	store word via A stack[A]:=B; RP:=RP-2
0 0 0 3 6 2	LDA	load double via A RP:=RP+1; cc(BA:=stack[B:B+1])
0 0 0 3 6 3	SDA	store double via A stack[A:A+1]:=CB; RP:=RP-3;
0 0 0 3 6 4	LBA	load byte via A ccb(A:=bytedest(A))
0 0 0 3 6 5	SBA	store byte via A bytedest(A):=B; RP:=RP-2
0 0 0 3 6 6	CDG	count duplicate words C=buffer address B=buffer size A=duplicate count  while B<>0 and stack[C]=stack[C-1] do {A:=A+1; B:=B-1; C:=C+1}
0 0 0 3 6 7	DFG	deposit field in memory 'Optionally Indivisible On'; cc(stack[A]:= (stack[A] & ~B)   (C & B)); 'Optionally Indivisible Off'; RP:=RP-3
0 0 0 3 7 1	SOPx	Software options: always present.  RP := RP + 2; B := -1; A := I - 2; cc(A);
0 0 0 3 7 7	SOPx	

**Table C-2. Instruction Definitions** (page 26 of 38)

0 0 0 4 0 3	TOPT	test for option - always present  A=option #  N:=0; Z:=0; RP:=RP-1
0 0 0 4 0 5*	FRST	firmware reset reset and stop instruction execution
0 0 0 4 0 6	LBX	load byte extended ccb(B:=bxmem[BA]); RP:=RP-1
0 0 0 4 0 7	SBX	store byte extended bxmem[BA]:=C; RP:=RP-3
0 0 0 4 1 0	LWX	load word extended c(B:=xmem[BA]); RP:=RP-1
0 0 0 4 1 1	SWX	store word extended xmem[BA]:=C; RP:=RP-3
0 0 0 4 1 2	LDDX	load double extended cc(BA:=xmem[BA:BA+3])  [ See Note 19: Sometimes Atomic ]
0 0 0 4 1 3	SDDX	store double extended xmem[BA:BA+3]:=DC; RP:=RP-4  [ See Note 19: Sometimes Atomic ]
0 0 0 4 1 4	LQX	load Quad extended RP:=RP+2; cc(DCBA:=xmem[DC:DC+7]) [ See Note 7: NOT ATOMIC ]
0 0 0 4 1 5	SQX	store Quad extended xmem[BA:BA+7]:=FEDC; RP:=RP-6 [ See Note 7: NOT ATOMIC ]
0 0 0 4 1 6	DFX	deposit field extended 'Optionally Indivisible On'; cc(xmem[BA]:=(xmem[BA] & ~C   (D & C))); RP:=RP-4 'Optionally Indivisible Off'  [ See Note 1: CALLABLE PRIV ]
0 0 0 4 1 7	MVBX	move bytes extended ED=destination address CB=source address A=byte count  while A<>0 do {bxmem[ED]:=bxmem[CB]; ED:=ED+1; CB:=CB+1; A:=A-1}; RP:=RP-5

**Table C-2. Instruction Definitions (page 27 of 38)**

0 0 0 4 2 0	MBXR	move bytes extended reverse ED=destination address CB=source address A=byte count  while A<>0 do {bxmem[ED]:=bxmem[CB]; ED:=ED-1; CB:=CB-1; A:=A-1}; RP:=RP-5
0 0 0 4 2 1	MBXX	move bytes extended and checksum F=initial xsum ED=destination address CB=source address A=byte count  while A<>0 do {bxmem[ED]:=t:=bxmem[CB]; F:=F xor t; ED:=ED+1; CB:=CB+1; A:=A-1}; RP:=RP-5
0 0 0 4 2 2	CMBX	compare bytes extended ED=destination address CB=source address A=byte count  N:=0; Z:=1; while Z and A<>0 do {cc(bxmem[ED]:bxmem[CB]); if Z then {A:=A-1;ED:=ED+1; CB:=CB+1}}; RP:=RP-5
0 0 0 4 2 3*	CRAX	convert relative to absolute extended address  BA = extended address  'Indivisible On' if B.<0> = 0 then {xa := BA; if B.<0:12> = 0 then {xa.<0:14> := case B.<13:14> of {0 ; 1 ; CCSEG[csegx].<0:14>; CCSEG[0].<0:14>; }; }; @vSEG := CPDST[ xa.<1:6> ] ; @page := vSEG[xa.<7:14>].<0:30>^0 <sup>1</sup> ]; p := xa.<15:31> '/' FRAMESZ; if p >= page.NATPAGES then IFAIL( No_Segment ); B.<0:14> := page.ABSSEG;  'Indivisible Off'

**Table C-2. Instruction Definitions (page 28 of 38)**

0 0 0 4 4 2*	RPT	read process timer 'Indivisible On'; RP := RP + 2; if not 'trusted copy of DS' then BA := PTIME + (TIMER) + (10000 * INTA.<13>) else BA := PTIME 'Indivisible Off'
0 0 0 4 4 3*	SPT	set process timer 'Indivisible On'; if not DS then PTIME := BA - TIMER - (INTA.<13> * 10000) else PTIME := BA; RP := RP - 2 'Indivisible Off'
0 0 0 4 4 4	SCS	set code segment  BA=byte address in current code  BA := BA + CCSEG[ csegx ]
0 0 0 4 4 5*	LQAS	load SG quad via A RP:=RP+3; cc(DCBA:=sysstack[D:D+3])
0 0 0 4 4 6*	SQAS	store SG quad via A sysstack[A:A+3]:=EDCB; RP:=RP-5
0 0 0 4 5 1	BPT	instruction breakpoint trap - Nonaccelerated Only  'unconditionally interrupt via SIV #19';
0 0 0 4 5 2*	BCLD	bus cold load simulate a bus cold load
0 0 0 4 5 3*	TPEF	test parity error freeze circuits NOP
0 0 0 4 5 4*	SCMP	set code "map"  A.<0:6> - internal procedure label A.<7:15> - XEP/PEP index  if A.<0:6> = 0 then {A.<0> := CS; A.<1> := LS; A.<2:6> := CSPACEINDEX } else if A.<0:6> = %133 then ! external call {@vSEG := CPDST[ CCSEG[ csegx ].<1:6> ]; @vseg := @vseg.<0:30>^01; @page := vSEG[ CCSEG[ csegx ].<7:14> ]; i := page.TNSSEGSZ*1024-1; A := code[i-A.<7:15>] } }

[ See Note 1: CALLABLE PRIV ]

**Table C-2. Instruction Definitions** (page 29 of 38)

0	0	0	4	5	6*	DDTX	DDT request
							A.<12:15>=request
							%004 = Enable XRAY sampler interrupts - TXP,VLX %005 = Disable XRAY sampler interrupts- TXP,VLX
							%020-%037 =Set diag flags with RUN bit-VLX only
							issue DDT request; if timeout then {N:=1; Z:=0} else {N:=0; Z:=1}; RP:=RP-1
0	0	0	4	6	1*	RUS	read micro state information RP:=RP-5
0	0	0	4	7	1	ESE	extensible stack entry
							A: expected-arguments word count
							if (xmem[Lx-\$XADR(3)]+A)=0 then RP := 7 else {cc(RP-1); DPCL(sysstack[%171]); RP := 7 cc and k now undefined }
							[ See Note 1: CALLABLE PRIV ]
0	0	0	5	0	0*	XADD	XRAY counter add
							D=parameter CB=ext addr A=counter offset
							'Indivisible On'; t := xmem2[CB]; if t.<0> then {a := t + \$XADR(A); ! add parameter xmem2[a]:=xmem2[a] + \$DBL(D); xmem[t] := 0; }
							'Indivisible Off'; RP:=RP-4;
							** The counters must be present in memory

**Table C-2. Instruction Definitions** (page 30 of 38)

0 0 0 5 0 1*	XSST	XRAY Set State
<pre> CB=ext addr A=counter offset  'Indivisible On'; t := xmem2[CB]; if t.&lt;0&gt; then {a := t + \$XADR(A); ! set state if xmem[a]=0 then {xmem[a]:=1; xmem[t] := 0; a:=a+2; clock:=sysstack[%350:%353] +microsecond counter; xmem4[a]:=xmem4[a]-clock } }; 'Indivisible Off'; RP:=RP-3;  ** The counters must be present in memory [ See Note 3: CLOCK ADDR  ] </pre>		
0 0 0 5 0 2*	XRST	XRAY Reset State
<pre> CB=ext addr A=counter offset  'Indivisible On'; t := xmem2[CB]; if t.&lt;0&gt; then {a := t + \$XADR(A); ! reset state if xmem[a]=1 then {xmem[a]:=0; xmem[t] := 0; a:=a+2; clock:=sysstack[%350:%353] + microsecond counter; xmem4[a]:=xmem4[a]+clock } }; 'Indivisible Off';  ** The counters must be present in memory [ See Note 3: CLOCK ADDR  ] </pre>		

**Table C-2. Instruction Definitions (page 31 of 38)**

0 0 0 5 0 3*	XIST	XRAY increment state
<pre> CB=ext addr A=counter offset  'Indivisible On'; t := xmem2[CB]; if t.&lt;0&gt; then {a := t + \$XADR(A); ! increment state if xmem[a]&lt;&gt;0xFFFF then {t:=xmem[a]:= xmem[a]+1; if xmem[a-2]'&lt;'t then xmem[a-2] := t; a:=a+2; clock:=sysstack[%350:%353] + microsecond counter; xmem4[a]:=xmem4[a] -clock xmem[t] := 0; } }; 'Indivisible Off'; RP:=RP-3;  ** The counters must be present in memory [ See Note 3: CLOCK ADDR ] </pre>		
0 0 0 5 0 4*	XDST	XRAY Decrement state
<pre> CB=ext addr A=counter offset  'Indivisible On'; t := xmem2[CB]; if t.&lt;0&gt; then {a := t + \$XADR(A); ! decrement state if xmem[a]&lt;&gt;0 then {xmem[a] := xmem[a]-1; xmem[t] := 0; a:=a+2; clock:=sysstack[%350:%353] + microsecond counter; xmem4[a]:=xmem4[a] +clock } }; 'Indivisible Off'; RP:=RP-3; ** The counters must be present in memory [ See Note 3: CLOCK ADDR ] </pre>		
0 0 0 5 0 5C	RTIM	Read System Time
<pre> RP := RP + 4; DCBA := CLOCK + 'micro second counter';  CLOCK = sysstack[ %350:%353 ] [ See Note 3: CLOCK ADDR ] </pre>		

**Table C-2. Instruction Definitions** (page 32 of 38)

0	0	0	5	2	6*	CAFL	Cache Flush
Inputs							
A - flags							
A.<0:13> = reserved							
A.<14> = flush data cache							
A.<15> = flush instruction cache							
CB - byte count							
ED - extended address							
'Indivisible On';							
vp te := RSPT(ED);							
if K then							
IFAIL( NO_Segment );							
K := 0;							
'flush cache( A.<15:14>, vp te ) for CB bytes;							
'Indivisible On';							
RP := RP - 5;							
[ See Note 10: TNS INSTRUCTION FAILURE ]							
0	0	1	-	-	-	CMPI	compare immediate
cc(A:imm); RP:=RP-1							
0	0	2	-	-	-	ADDS	add to S
Sx:=Sx+\$DBL(imm*2);							
if Sx '>=' %200000 then 'STACK OVERFLOW';							
0	0	3	-	-	-	LADI	logical add immediate
ccl(A:=A+'imm)							
0	0	4	0--	-	-	ORRI	or right immediate
cc(A:=A   I.<8:15>)							
0	0	4	4--	-	-	ORLI	or left immediate
cc(A:=A   (I.<8:15>'<<'8))							
0	0	5	-	-	-	LDLI	load left immediate
RP:=RP+1;							
cc(A:=imm rotate 8)							
0	0	6	-	-	-	ANRI	and right immediate
cc(A:=A&imm)							
0	0	7	-	-	-	ANLI	and left immediate
cc(A:=A&(imm rotate 8))							
1	0	0	-	-	-	LDI	load immediate
RP:=RP+1; cc(A:=imm)							
1	0	0xx	-	-	-	LDXI	load x immediate
cc(X:=imm)							
1	0	4	-	-	-	ADDI	add immediate
ccn(A:=A+imm)							
1	0	4xx	-	-	-	ADXI	add X immediate
ccn(X:=X+imm)							
I	1	0	0--	-	-	BIC	branch if carry
if K then branch							
I	1	1	0--	-	-	BGTR	branch if greater
if ~(N   Z) then branch							
I	1	2	0--	-	-	BEQL	branch if equal
if Z then branch							



**Table C-2. Instruction Definitions** (page 33 of 38)

I 1	3	0--	-	-	BGEQ	branch if greater or equal if ~ N then branch
I 1	4	0--	-	-	BLSS	branch if less if N then branch
I 1	5	0--	-	-	BNEQ	branch if not equal if ~ Z then branch
I 1	6	0--	-	-	BLEQ	branch if less or equal if N   Z then branch
I 1	7	0--	-	-	BNOC	branch no carry if ~ K then branch
I 1	0	4--	-	-	BUN	branch unconditional branch
I 1	0xx4	--	-	-	BOX	branch on X if X<A then {X:=X+1; branch} else RP:=RP-1
I 1	4	4--	-	-	BAZ	branch on A zero if A=0 then branch; RP:=RP-1
I 1	5	4--	-	-	BANZ	branch on A nonzero if A<>0 then branch; RP:=RP-1
I 1	6	4--	-	-	BNOV	branch if no overflow if ~ V then branch
I 1	7	4--	-	-	BSUB	branch to subroutine xmem[Sx:=Sx+2]:=TNSP; branch if Sx >= %200000 then 'stack overflow'
I 2	0xx0	--	-	-	LWP	load word from program RP:=RP+1; cc(A:=code[branch address+X])
I 2	0xx4	--	-	-	LBP	load byte from program RP:=RP+1; adr:=(if indirect then code[dba] else 0) +dba'<<'1+X; A:=codeb[ dba.<0>^(16 zeroes) + adr]; ccb(A) [Note: adr and dba are truncated to 16 bits.]
0 2	4	n	r	c	PUSH	push to stack xmem[Sx+2:Sx+c*2+2] :=R[(r-c)mod 8:r]; RP:=n; Sx:=Sx+(c+1)*2 if Sx >= %200000 then 'stack overflow'
0 2	5	0--	-	-	RSUB	return from subroutine  TNSP:=xmem[Sx]; Sx:=Sx-\$XADR(I.<8:15>) if P is register-exact point then use Accelerated Mode else use Nonaccelerated Mode
1 2	4	n	r	c	POP	pop from stack R[(r-c)mod 8:r] :=xmem[Sx-c*2:Sx]; RP:=n; Sx:=Sx-2*c-2

**Table C-2. Instruction Definitions (page 34 of 38)**

1	2	5	0	--	-	-	EXIT	exit procedure	<pre> xmap( stackb[Lx-2] &amp; %4437 ); (Sx,P,ENV,Lx):=(   Lx-I.&lt;8:15&gt;*2,   stackb[Lx-4],   (stackb[Lx-2])&amp;ENV&amp;%073000     stack[Lx-2]&amp;%104740   ENV&amp;%37,   \$XADR(stackb[Lx])); if P is register-exact point then use Accelerated Mode else use Nonaccelerated Mode if ENV.&lt;0&gt; then Instruction Breakpoint if ENV.V and ENV.T then overflow trap </pre>
0	2	5	4	-	-		LWXX	load word extended indexed	cc(A:=xmem[ (A^0 <sup>16</sup> )>>15+xbase])
0	2	6	4	-	-				
0	2	5	5	-	-		SWXX	store word extended indexed	xmem[ (A^0 <sup>16</sup> )>>15+xbase]:=B;
0	2	6	5	-	-				RP:=RP-2
0	2	5	6	-	-		LBXX	load byte extended indexed	ccb(A:=bxmem[\$DBL(A)+xbase])
0	2	6	6	-	-				
0	2	5	7	-	-		SBXX	store byte extended indexed	bxmem[(\$DBL(A)+xbase]:=B;
0	2	6	7	-	-				RP:=RP-2
0	2	6	00	mssd	n		MOVW	move words	<pre> while A&lt;&gt;0 do {dest(C):=source(B);   A:=A-1; B:=B+movestep;   C:=C+movestep}; RP:=n </pre>
0	2	6	02	mssd	n		COMW	compare words	<pre> N:=0; Z:=1; while Z and A&lt;&gt;0 do {cc(dest(C)':'source(B));   if Z then   {A:=A-1; B:=B+movestep;     C:=C+movestep}}; RP:=n </pre>
1	2	6	00	mssd	n		MOVB	move bytes	<pre> while A&lt;&gt;0 do {bytedest(C):=bytesource(B);   A:=A-1; B:=B+movestep;   C:=C+movestep}; RP:=n </pre>
1	2	6	02	mssd	n		COMB	compare bytes	<pre> N:=0; Z:=1; while Z and A&lt;&gt;0 do {cc(bytedest(C):   bytesource(B));   if Z then   {A:=A-1; B:=B+movestep;     C:=C+movestep}}; RP:=n </pre>

**Table C-2. Instruction Definitions (page 35 of 38)**

1	2	6	40mssd	n	SBW	scan bytes while while bytesource(B)<>0 and bytesource(B)=A do B:=B+movestep K:=bytesource(B)=0; RP:=n
1	2	6	42mssd	n	SBU	scan bytes until while bytesource(B)<>0 and bytesource(B)<>A do B:=B+movestep K:=bytesource(B)=0; RP:=n
0	2	7	-	-	PCAL	procedure call stack[Sx+2:Sx+6]:=           (Px.<15:30> ,(ENV & %177740)   CSPACEID ,Lx.<15:30>); t:=I.<7:15>; if ~ PRIV then {if t>=code[0] then {if t>=code[1] then priv trap; PRIV:=1 } } Lx:=Sx:=Sx+3*2; if Lx >= %200000 then 'stack overflow' P:=code[t]; RP:=7 if P is register-exact point then use Accelerated Mode else use Nonaccelerated Mode

**Table C-2. Instruction Definitions (page 36 of 38)**

1	2	7	-	-	-	XCAL	external call	<pre> stack[Sx+2:Sx+6]:= (Px.&lt;15:30&gt;                     , (ENV &amp; %177740)   CSPACEID                     , Lx.&lt;15:30&gt;); ! calculate CCSEG segment size @vSEG := CPDST[ (xa :=CCSEG[csegx] ).&lt;1:6&gt; ]; @page := vSEG[ xa.&lt;7:14&gt; ]; i:=page.TNSSEGSZ*%2000-1; t := code[i-I.&lt;7:15&gt;];  s.&lt;7&gt; := t.&lt;0&gt;;          ! CS s.&lt;4&gt; := t.&lt;1&gt;;          ! LS s.&lt;11:15&gt; := t.&lt;2:6&gt;; ! space index xmap( s ); m := t.&lt;1&gt;   t.&lt;0&gt;; p := t.&lt;7:15&gt;; if ~ PRIV then     {if p &gt;= mem[CCSEG[m],0] then       {'validate CCSEG with a trusted copy';        if p &gt;= mem[CCSEG[m],2]          then priv trap;         PRIV:=1       }     }; Lx:=Sx:=Sx+3*2; if Lx &gt;= %200000 then 'stack overflow' CS:=t.&lt;0&gt;; LS:=t.&lt;1&gt;; P:=code[t.&lt;7:15&gt;]; RP:=7 if P is register-exact point then use Accelerated Mode else use Nonaccelerated Mode </pre>
0	3	0	0	-	-	LLS	logical left shift	<pre> computeshiftcount(31); cc(A:=A'&lt;&lt;'shiftcount)  [ See Note 11: Single shifts &gt;= 32 ] </pre>
0	3	0	1	-	-	LRS	logical right shift	<pre> computeshiftcount(31); cc(A:=A'&gt;&gt;'shiftcount)  [ See Note 11: Single shifts &gt;= 32 ] </pre>
0	3	0	2	-	-	ALS	arithmetic left shift	<pre> computeshiftcount(31); cc(A:=A &lt;&lt; shiftcount)  [ See Note 13: Single Shift Counts &gt;= 32 ] [ See Note 17: Arith Left Shift Overflow ] </pre>
0	3	0	3	-	-	ARS	arithmetic right shift	<pre> computeshiftcount(31); cc(A:=A&gt;&gt;shiftcount)  [ See Note 13: Single Shift Counts &gt;= 32 ] </pre>

**Table C-2. Instruction Definitions (page 37 of 38)**

1 3 0 0 - -	DLLS	double logical left shift computeshiftcount(255); cc(BA:=BA'<<'shiftcount)	[ See Note 14: Double Shift Counts >= 256 ]
1 3 0 1 - -	DLRS	double logical right shift computeshiftcount(255); cc(BA:=BA'>>'shiftcount)	[ See Note 14: Double Shift Counts >= 256 ]
1 3 0 2 - -	DALS	double arithmetic left shift computeshiftcount(255); cc(BA:=BA <<shiftcount)	[ See Note 14: Shift Counts >= 256 ] [ See Note 17: Arith Left Shift Overflow ]
1 3 0 3 - -	DARS	double arithmetic right shift computeshiftcount(255); cc(BA:=BA>>shiftcount)	[ See Note 14: Double Shift Counts >= 256 ]
I 3 0xx - - -	LDX	load X cc(X:=word)	[ See Note 12: SG Address in Nonpriv Mode ]
I 3 4xx - - -	NSTO	nondestructive store wordx:=A	[ See Note 12: SG Address in Nonpriv Mode ]
I 4 0xx - - -	LOAD	load RP:=RP+1; cc(A:=wordx)	[ See Note 12: SG Address in Nonpriv Mode ]
I 4 4xx - - -	STOR	store wordx:=A; RP:=RP-1	[ See Note 12: SG Address in Nonpriv Mode ]
I 5 0xx - - -	LDB	load byte RP:=RP+1; ccb(A:=bytex)	[ See Note 12: SG Address in Nonpriv Mode ]
I 5 4xx - - -	STB	store byte bytex:=A.<8:15>; RP:=RP-1	[ See Note 12: SG Address in Nonpriv Mode ]
I 6 0xx - - -	LDD	load double RP:=RP+2; cc(BA:=dwordx)	[ See Note 7: NOT ATOMIC ] [ See Note 12: SG Address in Nonpriv Mode ]
I 6 4xx - - -	STD	store double dwordx:=BA; RP:=RP-2	[ See Note 7: NOT ATOMIC ] [ See Note 12: SG Address in Nonpriv Mode ]

**Table C-2. Instruction Definitions (page 38 of 38)**


---

I 7 0xx - - -	LADR	load address RP:=RP+1; A:=address.<15:30>+X  [ See Note 12: SG Address in Nonpriv Mode ]
I 7 4xx - - -	ADM	add to memory 'Optionally Indivisible On' ccn(wordx:=wordx+A); RP:=RP-1 'Optionally Indivisible Off' [ See Note 12: SG Address in Nonpriv Mode ]
I 7 4xx 0 4xx -*	ADM	add to SG memory 'Indivisible On'; ccn(wordx:=wordx+A); RP:=RP-1 'Indivisible Off'

---

# Compatibility Notes

The following list defines notes referred to in [Table C-2](#) on page C-11. These notes identify significant differences in the operation of an instruction in the NonStop (RISC-based) processors as compared to the operation in earlier TNS (CISC-based) processors.

1. **CALLABLE PRIV**  
These are nonprivileged instructions that require access to privileged state or memory. The implementation temporarily becomes privileged (and back) if necessary.
2. **PRIV ONLY**  
Nonprivileged accesses to SG in the NonStop processor terminate with an INSTRUCTION FAILURE. On TNS processors, such accesses simply modified short address space 0 instead.
3. **CLOCK ADDR**  
The TNS clock address of sysstack[ %103 : %106 ] is sysstack[ %350 : 353 ] in the NonStop processor.
4. **OVERFLOW RESULTS UNDEFINED**  
Results left after an overflow in the NonStop processor are not compatible with TNS overflow results on this instruction. They are undefined at the time of the exception and also after the instruction is finished.
5. **OVFL TRAP RESULTS UNDEFINED**  
Register state at the time of the overflow exception is undefined and will vary depending on the execution mode.
6. **OVERFLOW TNS COMPATIBLE**  
Overflow results after the instruction is finished are defined to be compatible with TNS processors.
7. **NOT ATOMIC**  
These instructions do not load or store all of their data without intervening interrupts. This is different from the operation in TNS processors, which sometimes were able to load or store atomically, and sometimes not, depending on the memory alignment of the data.
8. (Not used.)
9. **CME HANDLING**  
CME handling requires that INT32s be written into memory as aligned INT32s instead of as INT16s or unaligned INT32s.
10. **TNS INSTRUCTION FAILURE**  
These instructions are pseudoinstructions for the NonStop processor. TNS processors do not use these instructions and attempts to do so will cause an INSTRUCTION FAILURE interrupt.

11. Single Shifts  $\geq 32$   
Single-word shift counts in the range [16 : 31] have the same effect as a shift count of 16.  
  
Single-word shift counts less than 0 or greater than 31 give undefined answers.
12. SG Address in Nonpriv Mode  
SG addresses are not allowed in nonprivileged mode, and cause an INSTRUCTION FAILURE interrupt. TNS processors simply redirect the SG address to short address space 0.
13. (Not used.)
14. Double Shift Counts  $\geq 256$   
Doubleword shift counts less than 0 or greater than or equal to 256 give undefined answers. Doubleword shift counts in the range [32 : 255] have the same net effect as a shift count of 32.
15. EMPY Greater Precision  
EMPY results have more precision than TNS processors for certain multiplications. The NonStop answer is correct but different from TNS results.
16. Unimplemented TNS Op  
These TNS instructions do not exist in the instruction set of the NonStop processor and generate an INSTRUCTION FAILURE 'Unimplemented TNS Opcode' exception at run time, if executed.
17. Arith Left Shift Overflow  
TNS arithmetic left shift instructions ALS and DALS do not allow the sign bit of the result to change. Thus a (%040000 < 1) becomes %000000 after the TNS arithmetic left shift.  
  
The arithmetic left instructions behave this way in the NonStop processor only in nonaccelerated mode. The Accelerator treats all left shifts as logical left shifts, in which the sign bit is included in the bits being shifted. The result is the same bit pattern, using either kind of shift, in all normal nonoverflow cases where the operand was arithmetically small enough to avoid loss of significant bits when shifted. The result's sign bit is undefined in overflow cases.  
  
As on TNS processors, overflow cases of shift instructions never set V and never cause overflow traps.
18. Range Error  
On a range error in the NonStop processor, CC and R[7] differ from values given in TNS processors.
19. Sometimes Atomic  
This instruction is atomic if the doubleword is aligned.



---

---

---

---

---

# Glossary

**3-phase.** Describes a single power source with three output phases (A, B, and C). The phase difference between any two of the three phases or currents is 120 degrees.

**3860 ATM 3 ServerNet adapter (ATM3SA).** See [ATM 3 ServerNet adapter \(ATM3SA\)](#).

**3861 Ethernet 4 ServerNet adapter (E4SA).** See [Ethernet 4 ServerNet adapter \(E4SA\)](#).

**3862 Token-Ring ServerNet adapter (TRSA).** See [Token-Ring ServerNet adapter \(TRSA\)](#).

**3863 Fast Ethernet ServerNet adapter (FESA).** See [Fast Ethernet ServerNet adapter \(FESA\)](#).

**3865 Gigabit Ethernet ServerNet adapter (GESA).** See [Gigabit Ethernet ServerNet adapter \(GESA\)](#).

**4619 disk drive.** An 18-gigabyte, 15,000-rpm, small computer system interface (SCSI) disk drive for HP NonStop™ S-series servers running G06.06 and later software release version updates (RVUs). The 4619 disk drive can coexist and operate with lower-capacity or lower-speed drives in the same storage subsystem module.

**4637 disk drive.** A 36-gigabyte, 10,000-rpm, small computer system interface (SCSI) disk drive for HP NonStop™ S-series servers running G06.06 and later software release version updates (RVUs).

**6740 ServerNet/FX adapter.** See [ServerNet/FX adapter](#).

**6742 ServerNet/FX 2 adapter.** See [ServerNet/FX 2 adapter](#).

**6760 ServerNet device adapter (ServerNet/DA).** See [ServerNet device adapter \(ServerNet/DA\)](#).

**6761 F-PIC.** See [fiber-optic plug-in card \(F-PIC\)](#).

**6762 S-PIC.** See [SCSI plug-in card \(S-PIC\)](#).

**6763 Common Communication ServerNet adapter (CCSA).** See [Common Communication ServerNet adapter \(CCSA\)](#).

**A.** See [ampere \(A\)](#).

**A0CINFO file.** A distribution subvolume (DSV) file that contains information about a product and each of its files, including product and file dependencies, how the files are used and where they are placed, and which type of processor the product runs on. Every product and software product revision (SPR) to be managed by the Distributed Systems Management/Software Configuration Manager (DSM/SCM) is distributed in a subvolume and that subvolume must contain the product's A0CINFO file.

**absolute pathname.** An Open System Services (OSS) pathname that begins with a slash (/) character and is resolved beginning with the root directory. Contrast with [relative pathname](#).

**AC.** See [alternating current \(AC\)](#).

**accelerated mode.** The operational environment in which Accelerator-generated RISC instructions execute. See also [TNS mode](#) and [TNS/R native mode](#).

**accelerated object code.** The RISC instructions that result from processing a TNS object file with the Accelerator program.

**accelerated object file.** The object file that results from processing a TNS object file with the Accelerator program. An accelerated object file contains the original TNS object code, the accelerated object code and related address map tables, and any binder and symbol information from the original TNS object file.

**Accelerator program.** A program that processes a TNS object file and produces an accelerated object file. Most TNS object code that has been accelerated runs faster on TNS/R processors than TNS object code that has not been accelerated. The Accelerator program (AXCEL) is run prior to running the accelerated linker, XLLINK.

**access mode.** The form of file access permitted for a user or process.

**ACL.** See [automatic cartridge loader \(ACL\)](#).

**ACS.** See [automated cartridge subsystem \(ACS\)](#).

**action.** An operation that can be performed on a selected resource.

**activation.** The operator action of putting software into use after the software has been applied from the activation package to the target system.

**activation package.** A set of files containing product files, operator instructions, and instructions for applying the software on the target system. It consists of a header file containing the activation instructions and file attributes, multiple data files, Distributed Systems Management/Software Configuration Manager (DSM/SCM) control information, and Event Management Service (EMS) events.

**AC transfer switch.** A component of an [HP NonStop™ Cluster Switch \(model 6770\)](#) that provides access to dual AC power sources and the ability to switch between the two sources if one fails. The AC transfer switch draws power from its primary power source as long as it is available. If the primary source fails, the AC transfer switch is switched to draw power from the secondary power source.

**adapter.** See [ServerNet adapter](#).

**adapter cable.** (1) A cable that connects components that have incompatible electrical interfaces. (2) For the ServerNet wide area network (SWAN) concentrator, one of four

types of cable that can connect any of the six 50-pin WAN ports to one of the supported electrical interfaces (RS-232, RS-449, X.21, or V.35).

**ADAPTER object type.** The Subsystem Control Facility (SCF) object type for all adapters attached to your system.

**address space.** The memory locations to which a process has access.

**ADE.** See [application development environment \(ADE\)](#).

**adjacent SP.** A service processor (SP) that is directly connected through the ServerNet fabrics to the enclosure of a specified SP.

**administrator.** (1) For an HP NonStop™ system, the person responsible for the installation and configuration of a software subsystem on a NonStop node. Contrast with [operator](#). (2) For an IBM system, the person responsible for the day-to-day monitoring and maintenance tasks associated with a software subsystem on an IBM node. (3) For a UNIX system, the owner of `/dev/console`. The administrator is responsible for the installation and configuration of all hardware and software within a node.

**ADP.** See [Automated Data Processing \(ADP\)](#).

**ALLPROCESSORS paragraph.** A required paragraph in the CONFTEXT configuration file that contains attributes defining the HP NonStop™ Kernel operating system image for all system processors. The ALLPROCESSORS paragraph follows the optional DEFINES paragraph.

**alternate path.** A path not enabled as the preferred path. An alternate path can become a [primary path](#) when a primary path is disabled.

**alternating current (AC).** An electric current having a waveform that regularly reverses in positive and negative directions. North American electrical power alternates 60 times/second (60 hertz). Contrast with [direct current \(DC\)](#).

**amperage.** Current-carrying capacity, expressed in amperes.

**ampere (A).** The unit of electrical current or rate of flow of electrons. One volt across one ohm of resistance causes a current flow of one ampere. A flow of one coulomb/second equals one ampere.

**ANSI.** The American National Standards Institute.

**APE.** See [Application Program Examiner \(APE\)](#).

**API.** See [application program interface \(API\)](#).

**appearance side.** The side of a system enclosure that contains, behind a door, disk customer-replaceable units (CRUs) and power monitor and control unit (PMCU) CRUs. The appearance side is the side opposite the service side. System enclosures are

typically arranged so that the appearance side is the most visible side. See also [service side](#).

**application binary interface (ABI).** The conventions used to call functions and access global or external data.

**application development environment (ADE).** A set of methods and tools that are used throughout the lifecycle of an application project to design, code, and manage that project.

**Application Program Examiner (APE).** A tool used to browse through TNS object files that have been accelerated by the Accelerator. APE displays MIPS RISC code in addition to TNS code.

**application program interface (API).** A set of services (such as programming language functions or procedures) that are called by an application program to communicate with other software components. For example, an application program in the form of a client might use an API to communicate with a server program.

**application-specific integrated circuit (ASIC).** A custom-built integrated circuit (IC) used to perform highly specialized functions.

**Apply.** The Distributed Systems Management/Software Configuration Manager (DSM/SCM) action of executing the instructions contained in an activation package, such as placing new software on the target system and taking a snapshot of the new target system.

**appropriate privileges.** In the Open System Services (OSS) environment, an implementation-defined means of associating privileges with a process for function calls or function call options that need special privileges.

**Archive.** A set of unstructured files used to collect the software received onto the host system. Files received as input are placed in the Archive, and attributes of the files are stored in the host database. The planner specifies the Archive location in the Configuration Manager profile, using the Archive and Database Maintenance Interface.

**Archive and Database Maintenance Interface.** A block-mode interface run by a database or system administrator at both the host system and target systems to perform Distributed Systems Management/Software Configuration Manager (DSM/SCM) maintenance functions.

**ASCII.** American Standard Code for Information Interchange. A single-byte code set that uses only 7 of the 8 bits in a byte to represent each character. The ASCII code set contains the uppercase and lowercase characters of the U.S. English alphabet, some punctuation symbols, the digits 0 through 9, and some symbols and control characters. Because of its limited characters, and because the 8th bit is sometimes used in ASCII programs as a utility bit, the ASCII code set is not appropriate for use in international software.

**ASIC.** See [application-specific integrated circuit \(ASIC\)](#).

**ASSIGN.** An HP Tandem Advanced Command Language (TACL) command you can use to associate a file name with a logical file of a program, or to assign a physical device to logical entities that an application uses.

**assign message.** Within Subsystem Control Facility (SCF), a message created by SCF for each ASSIGN command. A new process must request its assign message following receipt of the startup message. All assign messages set by the SCF ASSIGN command, plus the ones read from the HP Tandem Advanced Command Language (TACL) command interpreter, are passed to the new process.

**assumed object.** The object type or object name specified by a Subsystem Control Facility (SCF) ASSUME command. If an ASSUME command has been used to establish a default object type and fully qualified default object name, and if that object type and object name together refer to a valid object, then *object-spec* can be omitted entirely from an SCF command, and the command is applied to the object known as the assumed object.

**asynchronous wide area network (AWAN) servers.** A local area network (LAN)-based communications device that provides (1) asynchronous connections to terminals, printers, and terminal emulators for HP NonStop S-series and K-series servers; (2) remote-access disk operating system (DOS), Windows, and Macintosh platforms; (3) VT-to-6530 protocol conversion; and (4) dial-out connections for LAN-attached DOS, Windows, and Macintosh platforms.

**ATM3SA.** See [ATM 3 ServerNet adapter \(ATM3SA\)](#).

**ATM 3 ServerNet adapter (ATM3SA).** A ServerNet adapter that provides access to Asynchronous Transfer Mode (ATM) networks from an HP NonStop™ S-series server. The 3860 ATM3SA supports the ATM User-Network Interface (UNI) specification over a 155-megabit/second (Mbps) OC-3 Sonet (Synchronous Optical Network) connection.

**atomic.** Behaving as a single, indivisible operation. For example, an atomic write operation on a file cannot write data that is interleaved with data from another, concurrent write operation on that file.

**attachment.** A file that contains information that augments the information in an incident report.

**attribute.** (1) For the Subsystem Control Facility (SCF), a characteristic of an entity. For example, two attributes of a process might be its program file and its user ID. An attribute is sometimes called a modifier. (2) In OSM and TSM client interfaces, a data item associated with a system or cluster resource. All attributes can be viewed, and some can be modified.

**audit.** A Distributed Systems Management/Software Configuration Manager (DSM/SCM) activity initiated by the operator at a target system that updates the target database with the fingerprints of all the files in a selected set of target subvolumes (TSVs).

**authentication attributes.** Security attributes of a process that do not change unless a successful reauthentication occurs or the super ID changes them. For Open System Services (OSS) processes, the authentication attributes include the login name, real user ID, real group ID, authentication system (node name), and group list.

**authorization attributes.** Security attributes of a process that can change through use of functions such as `setuid()` (or of Guardian procedures such as `PROCESS_CREATE_`) without reauthentication. For Open System Services (OSS) processes, the authorization attributes include the effective user ID, saved-set user ID, saved-set group ID, user audit flags, and effective user name.

**authorization key.** A password required for logging on to a modem. If you plan to allow dial-outs to a service provider, you must specify the authorization key of the service provider's modem during configuration of the OSM or TSM Notification Director.

**automated cartridge subsystem (ACS).** A type of tape library. Also known as automated cartridge system.

**Automated Data Processing (ADP).** The term used in the FIPS PUB 94 document to refer to computerized data processing equipment that is installed inside a computer room.

**automatic cartridge loader (ACL).** A device that stores multiple cartridge tapes and loads them automatically, one at a time, into a tape drive.

**automatic configuration.** The automatic assignment of magnetic disk attributes to an internal disk drive when it is inserted into a slot. Also known as "plug and play."

**averaging.** A measurement method for determining the average value of alternating voltage and current waveforms. The averaging method involves sampling a waveform and averaging the samples over the period of one cycle.

**AWAN.** See [asynchronous wide area network \(AWAN\) servers](#).

**AXCEL.** The command used to invoke the Accelerator on a TNS/R system.

**back-end board (BEB).** A circuit board that translates fiber-optic signals from a 3216 controller or 6760 ServerNet device adapter into small computer system interface (SCSI) commands and information for a tape drive. The BEB is housed in a cage-like sheet-metal enclosure and plugs into one of the 50-pin SCSI ports on the back of a tape drive customer-replaceable unit (CRU).

**background process.** In the Open System Services (OSS) environment, a process that belongs to a background process group.

**background process group.** In the Open System Services (OSS) environment, a process group that is both:

- Not a [foreground process group](#)
- A member of a session that has a connection with a controlling terminal



**backout.** The Distributed Systems Management/Software Configuration Manager (DSM/SCM) action of making the last configuration applied to the target system inaccessible and replacing it with the previous configuration.

**backplane.** A board that has connectors, on one or both sides of the board, into which circuit board assemblies plug. Backplanes are located behind card cages.

**BACKUP.** A utility for the HP NonStop™ servers that creates a backup copy of one or more disk files on magnetic tape. See also [RESTORE](#).

**backup processor.** A processor running the HP NonStop™ Kernel operating system that communicates with the [primary processor](#), allowing the processors to remain independent. A component failure in one processor has no effect on any other processor.

**base computing platform.** The minimum software implementation that is the foundation for the X/Open [common applications environment \(CAE\)](#).

**base enclosure.** An enclosure that is placed on the floor and can have other enclosures stacked on top of it. A base enclosure is installed on a frame base. Contrast with [stackable enclosure](#).

**base profile.** In an X/Open compliant system, a minimum set of software components required to create a common applications environment.

**battery load.** The electrical current drain imposed on a battery.

**BEB.** See [back-end board \(BEB\)](#).

**BIC.** Backplane interconnect card. Not applicable to HP NonStop™ S-series servers. See [ServerNet adapter](#).

**BIND.** A program invoked during system generation that creates TNS object (file code 100) system code files and system library files.

**Binder.** A programming utility that combines one or more compilation units' TNS object code files to create an executable TNS object code file for a TNS program or library. Used only with [TNS object files](#).

**Binder region.** The region of a TNS object file that contains header tables for use by the Binder program.

**binding.** The operation of collecting, connecting, and relocating code and data blocks from one or more separately compiled TNS object files to produce a target object file.

**bit-synchronous.** A type of Open Systems Interconnection (OSI) Layer-2 protocol that uses synchronous transmission but does not require a character code to define terminal and line control sequences.

**block.** A grouping of one or more system enclosures that an HP NonStop™ S-series system recognizes and supports as one unit. A block can consist of either one processor enclosure, one I/O enclosure, or one processor enclosure with one or more I/O enclosures attached.

**blocked signal.** A programmatic signal that is currently in the pending signal mask of a process and, when generated, is not delivered to the process because of the signal mask setting. Some signals cannot be blocked.

**block special file.** In the Open System Services (OSS) environment, a device that is treated as a file for which all input or output must occur in blocks of data. Traditionally, such files are disk or tape devices. Block special files provide access to a device in a manner that hides the hardware characteristics of the device. Contrast with [character special file](#).

**bond.** A reliable connection that ensures the required electrical conductivity between conductive parts that must be electrically connected.

**bonded.** The mechanical interconnection of conductive parts to maintain a common electrical potential.

**bonding.** The permanent joining of conductive parts to form a path that ensures electrical continuity and the capacity to safely conduct any current likely to be imposed.

**bonding jumper.** See [main bonding jumper](#).

**boot.** A synonym for [load](#). Load is the preferred term used in this and other HP NonStop™ S-series system publications.

**BOOTP.** A protocol for providing initialization information to diskless nodes in an open network.

**BOOTPC.** See [BOOTP client \(BOOTPC\)](#).

**BOOTP client (BOOTPC).** A client provided as a Portable Silicon Operating System (pSOS) system product task in the essential firmware on each communications line interface processor (CLIP) in the ServerNet wide area network (SWAN) concentrator. BOOTPC tasks are also provided on the host system as the WANBoot process in the WAN subsystem.

**BOOTPD.** See [BOOTP daemon \(BOOTPD\)](#).

**BOOTP daemon (BOOTPD).** The BOOTP server. One BOOTPD runs as a Portable Silicon Operating System (pSOS) system product task in the essential firmware on each communications line interface processor (CLIP) in the ServerNet wide area network (SWAN) concentrator. BOOTPD tasks are also provided on the host system as the WANBoot process in the WAN subsystem.



**branch circuit.** The circuit conductors located between the equipment receptacles and the final overcurrent device in a power distribution panel (PDP) that protect the circuits.

**branded product.** A software product that is licensed by X/Open to carry the X/Open or UNIX trademark.

**branding process.** The activities that lead to the acceptance of a product by X/Open in accordance with its Trade Mark Licence Agreement.

**break condition.** An event indicator or sequence of data from a terminal or terminal emulator that requests interruption of an application program.

**bridge rectifier.** A full-wave rectifier with four elements, as in a bridge circuit. Alternating voltage is applied to one pair of opposite junctions, and direct voltage is obtained from the other pair of junctions.

**BSD.** Berkeley Software Distribution.

**built configuration.** A configuration revision for which a system image and activation package have been created.

**built-in command.** In the Open System Services (OSS) environment, a command that is implemented within the `/bin/sh` file. Some built-in commands are also available as separately executable files.

**bypass mechanism.** Equipment that permits switching from one power source to another. For example, a bypass mechanism on an [uninterruptible power supply \(UPS\)](#) would switch to an alternative power source (such as a standby power generator or commercial utility source) when maintenance must be performed on the UPS.

**byte-synchronous.** A type of Open Systems Interconnection (OSI) Layer-2 protocol that uses synchronous transmission techniques and requires a character code to define terminal and line control sequences. Data is always transmitted in a block.

**cabinet.** One or more modules of a system, housed together.

**cable channel.** A cable management conduit that protects the cables that run between two system enclosures in a double-high stack. Each system enclosure has two cable channels running vertically on its service side: one on the left-hand side of the enclosure, and one on the right-hand side of the enclosure.

**cable guidepost.** A cable management rod that routes cables exiting the upper enclosure in a double-high stack to prevent the cables from hanging down in front of the customer-replaceable units (CRUs) in the base enclosure. A cable guidepost extends from the base of each cable channel.

**cable support.** A piece of cable management hardware that secures system cables. The cable support attaches to the service side of a system enclosure near the bottom of the enclosure. Cable ties for securing system cables are threaded through the cable

support. The cable support also contains the group and module ID labels and the rear group service light-emitting diode (LED).

**cache (cache memory).** A small, fast memory holding recently accessed data designed to speed up subsequent access to the same data. Cache memory is built from faster memory chips than main memory, and it is most often used with process or main memory but also used in network data transfer to maintain a local copy of data.

**cached bindings.** A copy in virtual memory of the data pages containing symbolic references that were rebound when a loadfile was loaded. The cached bindings are associated with a library import characterization that characterizes the set of loadfiles to which the symbols were bound. If the same file is subsequently loaded in an equivalent environment in the same processor, the cached bindings can be reused. See [fastLoad](#).

**CAE.** See [common applications environment \(CAE\)](#).

**canonical input mode.** For an Open System Services (OSS) process, a terminal input mode in which data is not made available to the process until an entire logical line (delimited by a newline, EOF, or EOL character) is entered. This mode is sometimes called line mode or nontransparent mode. Contrast with [noncanonical input mode](#).

**CAP.** See [cartridge access port \(CAP\)](#).

**Carbon Copy.** A remote operations software application that enables a workstation in one location to access, through a modem, a workstation in another location. Carbon Copy is included with all system consoles, and service providers use it to dial in to system consoles at customer sites. See also [remote access](#).

**card cage.** A structure made up of slots that hold components such as customer-replaceable units (CRUs) and ServerNet adapters.

**carrier.** (1) A sheet-metal structure that allows a single-high ServerNet adapter to be installed in a ServerNet adapter slot designed for a double-high ServerNet adapter. (2) An electrical signal that carries data.

**cartridge.** See [optical disk cartridge](#).

**cartridge access port (CAP).** The component on the optical storage library (OSL) and the tape libraries supported on HP NonStop™ S-series systems where you insert cartridges into and remove cartridges from the library.

**caught signal.** A programmatic signal that is delivered to a process that has a signal-handling function for it. When the signal is caught, the process is interrupted and the signal-handling function executes.

**CBB.** See [common base board \(CBB\)](#).

**CCITT.** International Telegraph and Telephone Consultative Committee.

**CCSA.** See [Common Communication ServerNet adapter \(CCSA\)](#).

**CE.** Customer engineer. See [service provider](#).

**cell.** See [storage cell](#).

**central processing unit (CPU).** Historically, the main data processing unit of a computer. The HP NonStop™ servers have multiple cooperating processors rather than a single CPU. See also [processor](#).

**Challenge Handshake Authentication Protocol (CHAP).** An Internet-standard protocol for verifying encrypted passwords. CHAP is a security protocol that is implemented using [Point-to-Point Protocol \(PPP\)](#). The OSM and TSM Notification Director use CHAP to maintain security during dial-outs.

**channel.** An information route for data transmission. See also [ServerNet link](#).

**CHAP.** See [Challenge Handshake Authentication Protocol \(CHAP\)](#).

**character.** A sequence of one or more bytes representing a single character; used for the organization, representation, or control of data. A single-byte character consists of eight bits that represent a character. A multibyte character uses one or more bytes to represent a character. A wide character is a fixed-width character wide enough to hold any coded character supported by an implementation.

The ISO C standard defines the term multibyte character; a single-byte character is a special case of multibyte character.

**character set.** A finite set of characters (letters, digits, symbols, ideographs, or control functions) used for the organization, representation, or control of data. See also [code set](#).

**character special file.** In the Open System Services (OSS) environment, a device that is treated as a file for which all input or output must occur in character bytes. Traditionally, such files are interactive terminals, and the ISO/IEC IS 9945-1:1990 standard defines only the access to such terminal files. See also [terminal](#). Contrast with [block special file](#).

**chassis.** A single sheet-metal structure that houses one set of system components. In an HP NonStop S-series server, a chassis is part of a system enclosure but can also be mounted in any standard 19-inch rack.

**checksum.** A generic term, meaning to “add” together (although the definition of “add” need not be a “normal” arithmetic add) all of the data to produce a check “word.” See also [cyclic redundancy check \(CRC\)](#).

**child process.** A process created by another process. The creating process becomes the parent process of the new process. See also [parent process](#).

**CIIN.** A command file in the SYS<sub>nn</sub> subvolume that is read and executed by the startup HP Tandem Advanced Command Language (TACL) process after system load if the CIIN file is specified in the CONFTEXT file and enabled in the OSM or TSM Low-Level Link. |

**circuit breaker.** A device designed to open and close a circuit by nonautomatic means and to open the circuit automatically on a predetermined overcurrent without damage to itself.

**CISC.** See [complex instruction-set computing \(CISC\)](#).

**CISC processor.** An instruction processing unit (IPU) that is based on complex instruction-set computing (CISC) architecture.

**class.** A group of object-oriented data entities and the methods associated with that group. |

**Class-1 CRU.** A customer-replaceable unit (CRU) that probably will not cause a partial or total system outage if the documented replacement procedure is not followed correctly. Customers replacing Class-1 CRUs do not require previous experience with replacing HP NonStop™ S-series CRUs. However, for some CRUs, customers must be able to use the tools needed for the replacement procedure (which are common tools) and must protect components from electrostatic discharge (ESD).

**Class-2 CRU.** A customer-replaceable unit (CRU) that might cause a partial or total system outage if the documented replacement procedure is not followed correctly. Customers replacing Class-2 CRUs should have either three or more months of experience with replacing HP NonStop™ S-series CRUs or equivalent training. Customers must be able to use the tools needed for the replacement procedure and must protect components from electrostatic discharge (ESD).

**Class-3 CRU.** A customer-replaceable unit (CRU) that probably will cause a partial or total system outage if the documented replacement procedure is not followed correctly. Customers replacing Class-3 CRUs should have either six or more months of experience with replacing HP NonStop™ S-series CRUs or equivalent training. Customers must be able to use the tools needed for the replacement procedure, must protect components from electrostatic discharge (ESD), and must understand the dependencies involved in NonStop S-series CRU-replacement procedures, such as disk-path switching. Replacement by a service provider trained by HP is recommended.

**client.** A software process, hardware device, or combination of the two that requests services from a server. Often, the client is a process residing on a programmable workstation and is the part of an application that provides the user interface. The workstation client might also perform other portions of the application logic.

**client application.** An application that requests a service from a [server application](#). Execution of remote procedure calls is an example of a client application.

**client (of a loadable library).** A loadfile that uses functions or data from a library.

**CLIP.** See [communications line interface processor \(CLIP\)](#).

**cluster.** (1) A collection of servers, or nodes, that can function either independently or collectively as a processing unit. See also [ServerNet cluster](#). (2) A term used to describe a system in a Fiber Optic Extension (FOX) ring. More specifically, a FOX cluster is a collection of processors and I/O devices functioning as a logical group. In FOX nomenclature, the term is synonymous with system or node.

**cluster number.** A number that uniquely identifies a node in a Fiber Optic Extension (FOX) ring. This number is in the range 1 through 14. See also [node number](#).

**cluster switch.** See [HP NonStop™ Cluster Switch \(model 6770\)](#) and [HP NonStop™ ServerNet Switch \(model 6780\)](#).

**cluster switch enclosure.** An enclosure provided by HP for housing the subcomponents of an HP NonStop™ Cluster Switch, which include the ServerNet II Switch, the AC transfer switch, and the uninterruptible power supply (UPS). A cluster switch enclosure resembles, but is half the height of, a standard HP NonStop S-series system enclosure.

**cluster switch group.** Within an external ServerNet fabric, all of the [cluster switches](#) that belong to the same [cluster switch zone](#). A cluster switch group can consist of up to four 6780 switches, each representing one [cluster switch layer](#). All of the cluster switches that form a cluster switch group typically are installed in the same [cluster switch rack](#).

**cluster switch layer.** The topological cluster switch position within a [cluster switch group](#). Each cluster switch group consists of up to four layers, numbered 1 to 4 from bottom to top. A cluster switch layer consists of a pair of [cluster switches](#) (X and Y) and provides connections for up to eight ServerNet nodes. Layers within a group are interconnected by intragroup cables. When all four layers are present, the intragroup cables are configured as a [vertical tetrahedron](#). See also [cluster switch layer number](#).

**cluster switch layer number.** A number in the range 1 through 4 that identifies the position of a [cluster switch](#) within a cluster switch group. See also [cluster switch group](#).

**cluster switch logic board.** A circuit board that provides switching logic for the [HP NonStop™ ServerNet Switch \(model 6780\)](#). The logic board (LB) has a front panel for operator and maintenance functions and is a [Class-3 CRU](#).

**cluster switch rack.** A mechanical frame consisting of or based on a 19-inch rack that supports the hardware necessary for a [cluster switch group](#).

**cluster switch zone.** A pair of X-fabric and Y-fabric [cluster switch groups](#) and the ServerNet nodes connected to them. Up to three zones are possible. The zones, if more than one, are interconnected by interzone cables, with each [cluster switch layer](#) cabled separately from the other layers.

**CME.** See [correctable memory error \(CME\)](#).

**CMI.** See [Communications Management Interface \(CMI\)](#).

**code segment.** A segment that contains executable instructions of a program or library to be executed plus related information. Code segments can be executed and also accessed as read-only data but not written to by an application program. These read-only and execute-only segments are efficiently shared among simultaneous executions of that program or library. Therefore, they are read from disk but are never written back to disk. See also [TNS code space](#).

**code set.** Codes that map a unique numeric value to each character in a character set, using a designated number of bits to represent each character. Single-byte code sets use 7 or 8 bits to represent each character. The ASCII and ISO 646 code sets use 7 bits to represent each character in Roman-based alphabets; these code sets are very limited and are not appropriate for international use. The single-byte ISO 8859 code sets use 8 bits to represent each character and can therefore support Roman-based alphabets and many others including Greek, Arabic, Hebrew, and Turkish. Multibyte code sets represent characters that require more than one byte, such as East Asian ideographic characters.

**code space.** See [TNS code space](#).

**cold load.** A synonym for [system load](#) or [load](#) (in the case of single processor load). System load or load is the preferred term in HP NonStop™ S-series system publications.

**command.** A demand for action by or information from a subsystem or the operation demanded by an operator or application. A command is typically conveyed as an interprocess message from an application to a subsystem.

**command file.** An EDIT file that contains a series of commands and serves as a source of command input.

**common applications environment (CAE).** A computer environment in which applications can be ported across all X/Open branded products because of the use of international and industry standards. A CAE is an open system application development environment, an open system execution environment, or a combination of the two.

**common base board (CBB).** In modular customer-replaceable units (CRUs), the printed wiring assembly (PWA) that plug-in cards (PICs) are installed in.

**Common Communication ServerNet adapter (CCSA).** A ServerNet adapter that provides an HP NonStop™ S-series integration platform for Signaling System Number 7 (SS7) protocol communications.

**common mode.** Electrical interference that can be measured as a ground-referenced signal. In true common mode, a signal is common to all of the current-carrying conductors.

**common-mode transients.** Transients that appear between both inputs of a circuit and a common reference (such as ground).



**communications line.** A two-way link consisting of processing equipment, I/O devices, protocol conventions, and cables that connect a computer to other computers.

**communications line interface processor (CLIP).** The major programmable device within the ServerNet wide area network (SWAN) concentrator, providing link-level protocol and a software interface to the host. The CLIP stores and implements specific communications protocols.

**Communications Management Interface (CMI).** A utility used in D-series and earlier release version updates (RVUs) to make online changes to the configuration of ATP6100, CP6100, and EnvoyACP/XF communications subdevices. In G-series RVUs, CMI functions are performed by the Subsystem Control Facility (SCF).

**communications subsystem.** The combination of data communications hardware and software processes that function together as an integrated unit to provide services and access to wide and local area networks.

**Compaq TSM.** Identifies a client or server software component used to manage or service HP NonStop™ S-series servers. See also [Compaq TSM client software](#) and [Compaq TSM server software](#).

**Compaq TSM client software.** The component of the Compaq TSM package that runs on a system console. The TSM client software consists of the TSM Low-Level Link Application, the TSM Service Application, the TSM Notification Director Application, and the TSM EMS Event Viewer Application. See also [Compaq TSM server software](#).

**Compaq TSM Event Viewer.** A component of the Compaq TSM client software. The TSM Event Viewer lets you set up criteria to view Event Management Service (EMS) log files in several ways, enabling you to rapidly assess service problems.

**Compaq TSM Low-Level Link.** A component of the Compaq TSM client software. The TSM Low-Level Link enables you to communicate with an HP NonStop™ S-series server even when the HP NonStop Kernel operating system is not running. When the operating system is running, you usually communicate with the server using the TSM Service Application. See also [Compaq TSM Service Application](#).

**Compaq TSM package.** A software product for HP NonStop™ S-series servers that provides the information needed to perform functions such as querying resources and testing, provides notification of problems on the system, and allows local or remote access to the system for service and maintenance. The TSM package performs the same role as that of HP Tandem Maintenance and Diagnostic System (TMDS), Syshealth, and Remote Maintenance Interface (RMI) on earlier systems.

**Compaq TSM server software.** The component of the Compaq TSM package that runs on an HP NonStop™ S-series server. When the HP NonStop Kernel operating system is running, the TSM client software on a system console communicates with a server through the TSM server software. See also [Compaq TSM client software](#).

**Compaq TSM Service Application.** A component of the Compaq TSM client software. The TSM Service Application enables you to communicate with an HP NonStop™ S-series server when the HP NonStop Kernel operating system is running. When the operating system is not running, communication must take place using the TSM Low-Level Link. See also [Compaq TSM Low-Level Link](#).

**compiler extended-data segment.** A selectable segment, with ID 1024, created and selected automatically in many (but not all) TNS processes. Within this segment, the compiler automatically allocates global and local variables and heaps that would not fit in the TNS user data segment. A programmer must keep this segment selected whenever those items might be referenced. Any alternative selections of segments must be temporary and undone before returning.

**complex instruction-set computing (CISC).** A processor architecture based on a large instruction set, characterized by numerous addressing modes, multicycle machine instructions, and many special-purpose instructions. Contrast with [reduced instruction-set computing \(RISC\)](#).

**compliance.** The testing and verification process that precedes X/Open licensing.

**computer-room power center (CRPC).** The equipment that conditions and distributes facility power to computer-room equipment. The CRPC typically houses an electrostatically shielded isolation transformer, power distribution panels (PDPs), a main shunt-trip circuit breaker, and voltage indicators. Also referred to as a power distribution unit (PDU) or power distribution center.

**concentrator manager process (ConMgr).** A process provided as part of the wide area network (WAN) subsystem. The ConMgr process runs in each processor that supports WAN products and provides management functions to the WAN subsystem and WAN products, such as downloading data link control (DLC) tasks to the communications line interface processors (CLIPs) on the ServerNet wide area network (SWAN) concentrator and selecting the preferred path for the DLC tasks.

**conduit.** A tubular raceway, usually constructed of rigid or flexible metal, through which insulated power and ground conductors or data cables are run. Nonmetallic conduits, although available, are not recommended.

**CONFAUX file.** The auxiliary configuration file created by the Distributed Systems Management/Software Configuration Manager (DSM/SCM) tools. The CONFAUX file contains a list of the code files and system files that are needed to build the new HP NonStop™ Kernel operating system. HP recommends that you avoid making any changes to your CONFAUX file.

**CONFBASE file.** In G-series release version updates (RVUs), the basic system configuration database file, which is stored on the \$SYSTEM.SYS<sub>nn</sub> subvolume. See also [configuration file](#).



**CONFIG file.** In G-series release version updates (RVUs), the current system configuration database file, which is stored on the \$SYSTEM.ZSYSCONF subvolume. See also [configuration file](#).

**configuration.** (1) The arrangement of enclosures, system components, and peripheral devices into a working unit. (2) The definition or alteration of characteristics of an object.

**configuration file.** In G-series release version updates (RVUs), one of the following files: CONFBASE, CONFIG, one or more saved configuration files named CONF<sub>xyyy</sub>, and CONFSAVE. See also [system configuration database](#). In pre-G-series RVUs, the configuration file is either the OSCONFIG file used by the Configuration Utility Program (COUP) or the CONFTEXT file used during system generation.

**configuration planner.** The person who manages system configuration changes and software configuration changes. This person modifies the system configuration database for system configuration changes and creates a new operating system image for software configuration changes. See also [planner](#).

**configuration revision.** A planner-defined set of software products and related configuration information that the Distributed Systems Management/Software Configuration Manager (DSM/SCM) can activate on a target system. Multiple configuration revisions might exist on a target system. A configuration revision is made up of the product versions named in its software revision list, its HP NonStop™ Kernel operating system image, and the relevant profile items, such as the location of the target subvolumes on the target system. It is created by a Build request and is included in the activation package sent to the target system.

**configuration tag.** Each configuration tag identifies and configures the topology of a cluster switch and its unique position within the topology. The configuration tag defines which ServerNet node numbers the cluster switch supports.

**configuration utility process.** The \$ZCNF process that is the access process for the CONFIG file and starts and maintains the \$ZPM persistence manager process.

**Configuration Utility Program (COUP).** A utility used in D-series and earlier release version updates (RVUs) to make online changes to the configuration of devices and controllers. COUP is part of the Dynamic System Configuration (DSC) facility. In G-series RVUs, similar functions are performed by the Subsystem Control Facility (SCF).

**configured object.** A Subsystem Control Facility (SCF) object that exists at the time a subsystem completes its initialization process, or an SCF object that is brought into existence by a command issued through a subsystem management interface.

**CONFLIST file.** The output file produced during system generation, including error and warning messages.

**conformance.** Meeting the requirements of a specific standard.

**conformance document.** An implementor's document that must accompany software claiming conformance with a POSIX standard. The document specifies the behavior or other aspect of the software when the standard describes a behavior or aspect as implementation-defined.

**conformance statement questionnaire (CSQ).** A document that identifies how a product implements X/Open Specifications as defined in XPG Component/Profile Definitions. A CSQ exists for each branded product.

**conforming POSIX.1 application.** An application that is either an [ISO/IEC-conforming POSIX.1 application](#) or a [national-standards-body conforming POSIX.1 application](#).

**conforming POSIX.1 application using extensions.** An application that:

- Is a [conforming POSIX.1 application](#).
- Also uses features or facilities that are not described in ISO/IEC IS 9945-1:1990 (POSIX.1) but are consistent with the standard.
- Meets the documentation requirements of a conforming POSIX.1 application and documents its use of nonstandard features or facilities.

For example, an application using the `tdm_fork()` function could be a conforming POSIX.1 application using extensions.

**CONFSAVE file.** In G-series release version updates (RVUs), the automatically saved configuration database file, which is stored on the \$SYSTEM.ZSYSCONF subvolume. See also [configuration file](#).

**CONFTEXT file.** The configuration file used as input during system generation that contains a series of entries defining your HP NonStop™ Kernel operating system attributes. A G-series CONFTEXT file consists of one or two paragraphs: DEFINES (optional) and ALLPROCESSORS.

**CONF~~xyy~~ file.** In G-series release version updates (RVUs), a saved configuration database file created by the Subsystem Control Facility (SCF) and stored in the \$SYSTEM.ZSYSCONF subvolume. `xyy` is the number you entered as `xx.yy` in the SCF SAVE CONFIGURATION command (`xx` indicates the base version and `yy` indicates the subversion). See also [configuration file](#).

**ConMgr.** See [concentrator manager process \(ConMgr\)](#).

**connection.** (1) The path between two protocol modules that provides reliable stream delivery service. (2) For OSM and TSM software, the logical link established between the client software on a workstation and the server software on an HP NonStop™ S-series system after a logon sequence has been performed. There are two types of logical connections: service connections and low-level links.

**Connection view.** One of several views of a server available in the view pane of the Management window of the Compaq TSM package. The Connection view is a visual

representation of the connectivity among components within an enclosure. See also [Physical view](#).

**connectivity.** The ability of a system to transfer information between itself and a system from another vendor. Other vendors use the term “connectivity” to mean hardware compatibility. See also [interoperability](#).

**connector.** See [port](#).

**console message.** See [operator message](#).

**contiguous ground.** An insulated grounding conductor that extends from an equipment enclosure power receptacle to the final point of electrical service for the computer-room equipment, whether that final point is the main service entrance or the separately derived power source. In most instances, the final point of electrical service is an isolating transformer installed in the computer room.

**control and inquiry.** Those aspects of Subsystem Control Facility (SCF) object management related to the state or configuration of an object. Such aspects include actions that affect the state or configuration of an object, inquiries about the object, and commands pertaining to the session environment (for example, commands that set default values for the session).

**controller.** See [I/O controller](#) or [ServerNet addressable controller \(SAC\)](#).

**controlling process.** In the Open System Services (OSS) environment, the session leader that established the connection to the controlling terminal. The session leader stops being the controlling process when the corresponding terminal stops being the controlling terminal.

**controlling terminal.** In the Open System Services (OSS) environment, a terminal that might be associated with a [session](#). A session can have only one controlling terminal, and a controlling terminal can control only one session at a time. When a session has a controlling terminal, all the following are true:

- Certain character sequences entered from that terminal cause signals to be sent to all processes in the process groups of that session.
- Certain characters entered from that terminal might receive special treatment.
- Members of background process groups of the session are restricted from certain kinds of access to the controlling terminal.

**Coordinated Universal Time (UTC).** The standard measure of time from the beginning of the current [Epoch](#). UTC is sometimes called Universal Coordinated Time, CUT, or UCT; the standard appellation is abbreviated as UTC, an arbitrary ordering of the letters. UTC was formerly called Greenwich mean time (GMT).

**core dump file.** See [saveabend file](#).

**core file.** See [saveabend file](#).

**correctable memory error (CME).** An error caused by incorrect data at a particular memory location. The cause of the error is such that the error is automatically corrected by the system. Contrast with [uncorrectable memory error \(UCME\)](#).

**COUP.** See [Configuration Utility Program \(COUP\)](#).

**CPU.** See [central processing unit \(CPU\)](#).

***cpu, pin.*** In the Guardian environment, a number pair that uniquely identifies a process during the lifetime of the process, consisting of the processor (CPU) number and the process identification number (PIN). See also [PID](#).

**CRC.** See [cyclic redundancy check \(CRC\)](#).

**creation version serial number (CRVSN).** In the Open System Services (OSS) environment, a number assigned by a disk process when a file is created. The CRVSN is used by the disk process and the OSS name server process to verify that the correct file is accessed. The CRVSN is stored in the catalog entry for an OSS regular file and is passed to the disk process when a Data Definition Language (DDL) request that involves the file is made.

**critical load.** Equipment that must have an uninterruptible power input to prevent damage to the equipment or the facility, or injury to personnel.

**CRPC.** See [computer-room power center \(CRPC\)](#).

**CRU.** See [customer-replaceable unit \(CRU\)](#).

**CRVSN.** See [creation version serial number \(CRVSN\)](#).

**CSQ.** See [conformance statement questionnaire \(CSQ\)](#).

**current.** The movement of electrons caused by potential difference between two electromotive charge forces.

**current configuration file.** See [configuration file](#).

**current working directory.** In the Open System Services (OSS) environment, the directory used in pathname resolution of relative pathnames. A process always has a current working directory. See also [working directory](#).

**CUSTFILE.** An EDIT file included on every site update tape (SUT) as \$SYSTEM.Annnnnnn.CUSTFILE, where *nnnnnn* is the system serial number of the target system. The CUSTFILE contains information on the software products on the SUT, their related files, and the destination and use of each file. HP customizes information in the CUSTFILE for each customer's system.

**customer engineer (CE).** See [service provider](#).

**customer-installable system.** A system that does not require specially trained service providers to install.

**customer-replaceable unit (CRU).** A unit that can be replaced in the field either by customers or by qualified personnel trained by HP. CRUs are divided into the categories of Class 1, Class 2, and Class 3 according to the risk of causing a system outage if the documented replacement procedure is not followed correctly and how much CRU-replacement training or experience is advisable. See also [Class-1 CRU](#), [Class-2 CRU](#), [Class-3 CRU](#), and [field-replaceable unit \(FRU\)](#).

**cyclic redundancy check (CRC).** The most widely used error detection code for ensuring the integrity of transmitted data. The digits of the CRC are calculated by the sender for each block of data sent and recalculated by the receiver (it is a family of mathematical functions involving computing the quotient and remainder of a polynomial division). A CRC is a form of [checksum](#).

**daemon.** See [demon](#).

**dark site.** See [unattended site](#).

**data communications equipment (DCE).** Equipment that provides all the functions required to establish, maintain, and terminate a connection and provides the signal conversion and coding between the data terminal equipment (DTE) and telephone company lines or data circuits. A DCE is usually a modem.

**data link control (DLC).** A set of functions associated with Layer 2 of the Open Systems Interconnection (OSI) reference model. These functions are responsible for reliable communication between two physically connected nodes.

**data link control (DLC) task.** Tasks that support the equivalent to Layer 2 of the Open Systems Interconnection (OSI) reference model. Wide area network (WAN) DLC tasks execute in the ServerNet wide area network (SWAN) concentrator communications line interface processor (CLIP), and each WAN DLC task controls one line interface.

**data segment.** A virtual memory segment holding data. Every process begins with its own data segments for program global variables and runtime stacks (and for some libraries, instance data). Additional data segments can be dynamically created. See also [flat segment](#) and [selectable segment](#).

**data terminal equipment (DTE).** Equipment that constitutes the data source or data sink and provides for the communication control function protocol; it includes any piece of equipment at which a communication path begins or ends.

**data transparent.** Describes software that examines all eight bits of every data byte and that uses no bit in a data byte for its own purposes. Internationalized applications must be data transparent.

**dB.** See [decibel \(dB\)](#).

**dBm.** Decibels as referenced to a milliwatt. A unit of measure that establishes 0 dBm equal to 1 milliwatt. A negative value represents a decrease in power, and a positive value represents an increase in power. See also [decibel \(dB\)](#).

**DC.** See [direct current \(DC\)](#).

**DCE.** See [data communications equipment \(DCE\)](#).

**DCF.** See [dynamic configuration file \(DCF\)](#).

**DC power cable.** In system enclosures with power shelves, a cable that delivers DC power from the power shelf to a processor multifunction (PMF) customer-replaceable unit (CRU) or I/O multifunction (IOMF) CRU in that enclosure.

**decibel (dB).** A unit of measure used to express a relative difference in power. A negative value represents a decrease in power, and a positive value represents an increase in power.

**dedicated service LAN.** An Ethernet local area network (LAN) for use by only OSM and TSM applications. This LAN connects system consoles with the Ethernet ports on the processor multifunction (PMF) customer-replaceable units (CRUs) in group 01 of an HP NonStop™ S-series server. A dedicated LAN supports NonStop S-series servers and system consoles but does not support any other types of servers or workstations. See also [public LAN](#).

**DEFINE.** An HP Tandem Advanced Command Language (TACL) command you can use to specify a named set of attributes and values to pass to a process.

**DEFINES paragraph.** An optional paragraph in the CONFTEXT configuration file that contains one or more identifiers, each with its associated text string. The DEFINES paragraph, if used, precedes the ALLPROCESSORS paragraph.

**delta.** A method for connecting a 3-phase power source (or load) in a closed series loop with input (or output) connections made to each of the three junctions. The delta's physical arrangement resembles the delta character from the Greek alphabet.

**demon.** On a UNIX system, a process that runs continuously to provide a specific service for other processes. A demon does not have a controlling terminal and is not explicitly invoked. On an HP NonStop™ system, a demon runs in the Open System Services (OSS) environment and has an OSS process ID. See also [static server](#).

**destination ServerNet ID (DID).** A field in the ServerNet packet header indicating the intended destination for the packet.

**detailed report.** A complete listing of status or configuration information provided by the Subsystem Control Facility (SCF) STATUS or INFO command when you use the DETAIL option. Contrast with [summary report](#).



**device.** A computer peripheral or an object that appears to an application as such. See also [terminal](#).

**dial-out point.** A system console from which incident reports are sent to a service provider. Incident reports are sent only from system consoles defined as the primary and backup dial-out points (the primary and backup system consoles).

**DID.** See [destination ServerNet ID \(DID\)](#).

**DIMM.** See [dual inline memory module \(DIMM\)](#).

**direct current (DC).** Electric current that flows in only one direction. Contrast with [alternating current \(AC\)](#).

**direct jump area.** One of sixteen 256-megabyte portions of the 4-gigabyte virtual address space. A RISC jump instruction has the ability to jump directly to any location within its own direct jump area without having to use a far jump table.

**directory.** A type of Open System Services (OSS) special file that contains directory entries, which associate names with files. No two directory entries in the same directory have the same name.

**directory entry.** In the Open System Services (OSS) file system, an object that associates a filename with a file. Several directory entries can associate names with the same file. See also [link](#).

**directory loop.** In the Open System Services (OSS) file system, an error condition in which a directory is identified as its own parent directory.

**directory special file.** See [directory](#).

**directory stream.** In the Open System Services (OSS) file system, an object with an opaque data type. A process can sequentially read directory entries from a directory stream.

**directory tree.** A hierarchy of directories. In the Open System Services (OSS) environment, directories are connected to each other in a branching hierarchical fashion such that only one path exists between any two directories (if no backtracking occurs).

**disconnecting means.** A device, group of devices, or other means by which the conductors of a circuit can be disconnected from their source of supply.

**discovery.** For the OSM and TSM client software, the process of identifying the resources that exist on an HP NonStop™ S-series server. See also [incremental discovery](#) and [initial discovery](#).

**disk bootstrap.** A software entity residing on disk that is used to load the HP NonStop™ Kernel operating system image (OSIMAGE) into memory during a system load. A disk that contains the disk bootstrap is referred to as a bootable disk. The disk bootstrap is

placed on the disk either as part of a tape load or as a result of the SCF CONTROL DISK, REPLACEBOOT command.

**disk cache.** A temporary storage buffer into which data is read, retained, and perhaps updated before being written to disk, for more efficient processing.

**disk drive.** A device that stores and accesses data on a disk. There are two types of disk drives: magnetic and optical. Random access to addressable locations on a magnetic disk is provided by magnetic read/write heads. Random access to addressable locations on an optical disk is provided by a low-intensity laser. See also [volume](#).

**DISKGEN.** A system generation option that invokes the DISKGEN program to copy directly to disk those files necessary to generate an HP NonStop™ Kernel operating system. DISKGEN can be used instead of a system image tape (SIT).

**DISK object type.** The Subsystem Control Facility (SCF) object type for all disk devices attached to your system.

**disk volume.** See [volume](#).

**distributed system.** A system that consists of a group of connected, cooperating computers.

**Distributed Systems Management (DSM).** A set of tools used to manage HP NonStop™ S-series systems and Expand networks.

**Distributed Systems Management/Software Configuration Manager (DSM/SCM).** A graphical user interface (GUI)-based program that installs new software and creates a new HP NonStop™ Kernel operating system. DSM/SCM creates a new software revision and activates the new software on the target system.

**distribution subvolume (DSV).** A subvolume containing program files for a particular software product along with the software release version update (RVU) document (softdoc) file for that product. The format for a DSV name is *Ynnnnrrrr* or *Rnnnnrrrr*, where *nnnn* is the software product number and *rrr* is the base version identifier (such as D20) or software product revision (SPR) identifier (such as AAB).

**DLC.** See [data link control \(DLC\)](#).

**DNS.** See [Domain Name System \(DNS\)](#).

**DNS server.** A server that resolves hostnames to Internet protocol (IP) address mapping queries. These queries originate from either client computers, which are known as resolvers, or other [Domain Name System \(DNS\)](#) servers, which accounts for the distributed nature of DNS. See also [Network Information Service \(NIS\)](#).

**domain.** (1) In the Internet, a part of the naming hierarchy. Syntactically, a domain name consists of a sequence of names (labels) separated by periods (dots).



(2) In a NonStop S-series server, a pair of service processors, the associated router clouds, and the attached replaceable units.

(3) A set of objects over which control or ownership is maintained. Types of domains include power domains and service processor (SP) domains.

**Domain Name System (DNS).** A system that defines a hierarchical, yet distributed, database of information about hosts on a network. The network administrator configures the DNS with a list of hostnames and Internet protocol (IP) addresses, allowing users of workstations that are configured to query the DNS to specify remote systems by hostnames rather than by IP addresses. DNS domains should not be confused with Windows NT networking domains. See also [DNS server](#), [Network Information Service \(NIS\)](#), and [ping](#).

**donor system.** The computer system you make smaller by removing enclosures, either to reduce the system or to add the removed enclosures to another [target system](#), using a process known as system reduction.

**double-high ServerNet adapter.** A ServerNet adapter that occupies an entire ServerNet adapter slot in an HP NonStop™ S-series server. Contrast with [single-high ServerNet adapter](#).

**double-high stack.** A stack that includes a base, a frame, and two system enclosures. Contrast with [single-high stack](#).

**double-wide plug-in card (PIC).** A large-form-factor [plug-in card \(PIC\)](#) that occupies two adjacent PIC slots within a customer-replaceable unit (CRU). See also [single-wide plug-in card \(PIC\)](#).

**download.** The process of transferring software from one location to another, where the transferring entity initiates the transfer.

**download line task.** Any task running under the Portable Silicon Operating System (pSOS) system product, such as a data protocol.

**downtime.** Time during which a computer system is not capable of doing useful work because of a planned or unplanned outage. From the end user's perspective, downtime is any time a needed application is not available.

**downward compatibility.** The ability of a requester to operate with a server of an earlier revision level. In this case, the requester is downward-compatible with the server and the server is upward-compatible with the requester. Contrast with [upward compatibility](#).

**drive.** See [disk drive](#), [optical disk drive](#), or [tape drive](#).

**dropout.** A voltage loss of very short duration (that is, milliseconds).

**DSC.** See [Dynamic System Configuration \(DSC\)](#).

**DSM.** See [Distributed Systems Management \(DSM\)](#).

**DSM/SCM.** See [Distributed Systems Management/Software Configuration Manager \(DSM/SCM\)](#).

**DSV.** See [distribution subvolume \(DSV\)](#).

**DTE.** See [data terminal equipment \(DTE\)](#).

**dual inline memory module (DIMM).** Small circuit boards carrying memory integrated circuits, with signal and power pins on both sides of the board. A DIMM is different from a single inline memory module (SIMM) in that the connections on each side of the module connect to different chips, whereas the connections on both sides of a SIMM connect to the same memory chip. This difference gives the DIMM a wider data path, as more modules can be accessed at once.

**dual-ported.** The capability of a ServerNet adapter or peripheral device to receive data and commands from two sources although only one source might have access at any particular moment.

**duplicate file descriptor.** In the Open System Services (OSS) file system, a file descriptor that refers to the same open file description as another file descriptor.

**dynamic configuration file (DCF).** An attachment file that is produced by the OSM and TSM client software and accompanied by an incident report. The DCF contains a snapshot of the system configuration, the state of the HP NonStop™ S-series server, and outstanding alarms at the time that the incident report was issued. The DCF is used by the service provider to avoid having to perform online discovery of the server over dial-up telephone lines.

**dynamic information.** Information that represents the set of resources that actually exist in the current configuration of an HP NonStop™ S-series server. Dynamic information is gathered from a server through the process of discovery. Contrast with [static information](#).

**dynamic-link library (DLL).** A collection of procedures whose code and data can be loaded and executed at any virtual memory address, with run-time resolution of links to and from the main program and other independent libraries. The same DLL can be used by more than one process. Each process gets its own copy of DLL static data. Contrast with [shared run-time library \(SRL\)](#). See also [TNS/R library](#).

**dynamic loading.** Loading and opening dynamic-link libraries under programmatic control after the program is loaded and execution has begun.

**dynamic process configuration.** Using Subsystem Control Facility (SCF) to configure a generic process to always start in a designated primary processor (that is, to be fault-tolerant).

**dynamic shared object (DSO).** See [dynamic-link library \(DLL\)](#).

**Dynamic System Configuration (DSC).** A utility used in D-series and earlier release version updates (RVUs) to make online changes to the configuration of devices and controllers. Its interactive utility is called the Configuration Utility Program (COUP). In G-series RVUs, similar functions are performed by Subsystem Control Facility (SCF).

**E4SA.** See [Ethernet 4 ServerNet adapter \(E4SA\)](#).

**earth ground.** The connection of the electrical-grounding conductors to a dependable, low-resistance contact with the soil.

**earth-grounding electrode.** An electrically conductive rod that is driven into soil, thus providing an earth-ground connection point for the electrical ground wiring in a building. A vertical steel column of a building, with its base sunk into soil, can also serve as an earth-grounding electrode.

**earth-grounding electrode system.** A grounding network created by bonding together the grounding means in a building (for example, underground metal water pipes, structural steel, and ground rods into the earth) and bonding them to the switchgear at the facility's main electrical service entrance.

**ECL.** See [emitter-coupled logic \(ECL\)](#).

**ECL plug-in card (PIC).** See [emitter-coupled logic \(ECL\) plug-in card \(PIC\)](#).

**ECL ServerNet cable.** See [emitter-coupled logic \(ECL\) ServerNet cable](#).

**EDIT file.** In the Guardian file system, an unstructured file with file code 101. An EDIT file can be processed by either the EDIT or PS Text Edit (TEDIT) editor. An EDIT file typically contains source program or script code, documentation, or program output. Open System Services (OSS) functions can open an EDIT file only for reading.

**effective group ID.** An attribute of an Open System Services (OSS) process that is used to determine permissions such as the file access allowed for the process. The effective group ID of a process is a group ID that contributes to the group access privileges of that process. The effective group ID of a process might be used to set the group ID of files created by that process. The effective group ID can be changed while the process runs.

**effective user ID.** An attribute of an Open System Services (OSS) process that is used to determine such permissions as the file access allowed for the process. The effective user ID of a process is the user ID that determines the owner access privileges of that process. The effective user ID of a process might be used to set the user ID of files created by that process. The effective user ID can be changed while the process runs.

**EIA.** Electronic Industries Association.

**electric utility.** The local utility service that, for a fee, supplies alternating-current (AC) power to businesses and residences.

**electromagnetic interference (EMI).** Forms of conducted or radiated interference that might appear in a facility as either normal or common-mode signals. The frequency of the interference can range from the kilohertz to gigahertz range. However, the most troublesome interference signals are usually found in the kilohertz to low megahertz range. At present, the terms electromagnetic interference and [radio frequency interference \(RFI\)](#) are usually used interchangeably.

**electrostatically shielded transformer.** A transformer that has a metallic shield placed between the primary and secondary windings. This shield diverts high-frequency signals to ground.

**electrostatic discharge (ESD) protection kit.** A kit containing an antistatic mat and a wriststrap with a cable and grounding clip. A service provider or customer wears the wriststrap while performing maintenance procedures inside an enclosure. The wriststrap and cable contain grounding wires so that when the grounding clip is attached to a metal object, such as the enclosure, the person wearing the wriststrap is grounded and any static electricity incurred during the procedure is discharged safely to the enclosure instead of to electrical components within the enclosure.

**ELF.** See [extended link format \(ELF\)](#).

**emergency power off (EPO).** Describes equipment used to automatically disconnect all electrical power to connected equipment if there is an emergency. A computer room's main EPO system shuts off all room equipment (except for lighting and fire-sensor equipment) if there is a fire. An equipment zone EPO shuts off power to all connected computer equipment if a power anomaly occurs.

**emergency power-off (EPO) connector.** A two-pin connector on the service side of an enclosure that allows an external signal to disable the batteries in the enclosure during emergency conditions. A cable is attached from the connector to a relay band or push button typically located near the door of a computer room. Pushing the EPO button removes power from all computer equipment in the room. The EPO connectors prevent the batteries from powering the server after power is removed. EPO capabilities are required in the United States when a server is installed in a computer room designed to comply with the special construction and fire protection provisions of the United States' national electrical code (or at other sites as required by local regulations.)

**EMI.** See [electromagnetic interference \(EMI\)](#).

**emitter-coupled logic (ECL).** A logic that expresses digital signals in differential negative voltage levels, from -8 volts to -1.8 volts. HP NonStop™ S-series servers containing ServerNet expansion boards (SEBs) use ECL ServerNet cables. An ECL plug-in card (PIC) allows the modular SEB (MSEB) and I/O multifunction (IOMF) 2 customer-replaceable unit (CRU) to use ECL ServerNet cables.

**emitter-coupled logic (ECL) plug-in card (PIC).** A plug-in card (PIC) for the modular ServerNet expansion board (MSEB) and I/O multifunction (IOMF) 2 customer-

replaceable unit (CRU) that supports the emitter-coupled logic (ECL) interface. See also [emitter-coupled logic \(ECL\)](#) and [plug-in card \(PIC\)](#).

**emitter-coupled logic (ECL) ServerNet cable.** A ServerNet cable that uses [emitter-coupled logic \(ECL\)](#). Before the modular ServerNet expansion board (MSEB) was introduced, ECL was the only ServerNet cable technology used by HP NonStop™ S-series servers. You can connect an ECL ServerNet cable directly to a ServerNet expansion board (SEB) or to an MSEB using an ECL plug-in card (PIC).

**empty directory.** In the Open System Services (OSS) file system, a directory that contains only an entry for itself and an entry for its parent directory.

**empty string.** In C and C++ programs, a character string that begins with a null character. This term is synonymous with “null string.”

**EMS.** See [Event Management Service \(EMS\)](#).

**EMS collector.** An Event Management Service (EMS) process to which subsystems report events.

**enclosure.** Similar to a cabinet in HP NonStop™ K-series systems. An enclosure can contain components of a system or a peripheral. Base enclosures are placed on the floor and can have other enclosures stacked on top of them. Stackable enclosures can be placed on top of other enclosures. See also [system enclosure](#) and [peripheral enclosure](#).

**enclosure interleaving.** On HP NonStop™ S-series systems, configuring a mirrored disk volume to use two separate system enclosures. For internal disk drives, the two disk drives of the mirrored volume can be in separate enclosures. For external disk drives, the adapters connected to the two disk drives of the mirrored volume can be in separate enclosures.

**endian.** Denotes the significance of byte 0 in a multibyte structure such as a word. NonStop servers are big-endian, where the most significant bit is contained in byte 0; Intel systems and HP AlphaServer OpenVMS and HP AlphaServer Tru64 UNIX systems are little-endian, where the least significant bit is contained in byte 0.

**environmental parameters.** Subsystem Control Facility (SCF) session parameters set by default or by using various SCF commands. The values associated with the environmental parameters can be examined using the ENV command.

**environment strings.** For an Open System Services (OSS) process, a vector of strings of the form *name = value* that contains information about the environment that the process runs in. Environment strings are accessible to the process and are inherited by its child processes.

**EPO.** See [emergency power off \(EPO\)](#).

**Epoch.** The period beginning January 1, 1970, at 0 hours, 0 minutes, and 0 seconds [Coordinated Universal Time \(UTC\)](#).

**EPO connector.** See [emergency power-off \(EPO\) connector](#).

**equipment grounding conductor.** The conductor used to connect the non-current-carrying metal parts of equipment, raceways, and other enclosures to the grounding electrode conductor at the facility's main service entrance or at the source of a separately derived power source.

**errno.** An external variable that contains the most recent error condition set by a C function.

**error number.** For the Subsystem Programmatic Interface (SPI), a value that can be assigned to a return token, or to the last field of an error token, to identify an error that occurred. SPI defines a small set of error numbers, but most error numbers are defined by subsystems.

**ESD kit.** See [electrostatic discharge \(ESD\) protection kit](#).

**ESP.** See [expansion service processor \(ESP\)](#).

**essential firmware.** Code in memory that is necessary for power-up initialization and communication with a host or device. Contrast with [nonessential firmware](#).

**Ethernet.** A local area network (LAN) that uses the carrier sense multiple access with collision detection (CSMA/CD) access method on a bus topology and is the basis for the IEEE 802.3 standard.

**Ethernet 4 ServerNet adapter (E4SA).** A ServerNet adapter for Ethernet local area networks (LANs) that contains four Ethernet ports.

**Ethernet hub.** A multiport repeater typically supporting 10Base-T cabling. Most hubs are connectors for 8 or 12 cables. Also referred to as a concentrator.

**Event Management Service (EMS).** A Distributed Systems Management (DSM) product that provides event collection, event logging, and event distribution facilities. EMS provides different event descriptions for interactive and programmatic interfaces, lets an operator or an application select specific event-message data, and allows for flexible distribution of event messages within a system or network.

**event message.** Text intended for a system operator that describes a change in some condition in the system or network, whether minor or serious. The change of condition is called an event. Events can be operational errors, notifications of limits exceeded, requests for actions needed, and so on. See also [operator message](#).



**Event Viewer Server Manager.** A persistent process that routes messages and data between the OSM or TSM Event Viewer on the system console and event server and summary server processes on the HP NonStop™ S-series server. An event server process retrieves events. A summary server is a persistent process that maintains a summary of all the events in a specified log file.

**exception handler.** A section of program code to which control is transferred when an exception occurs. The exception handler then determines what action should be taken.

**Expand line-handler process.** A process pair that handles incoming and outgoing Expand messages and packets. An Expand line-handler process handles direct links and also binds to other processes using the Network Access Method (NAM) interface to support Expand-over-X.25, Expand-over-FOX, Expand-over-ServerNet, Expand-over-TCP/IP, and Expand-over-SNA links. See also [Expand-over-ServerNet line-handler process](#).

**Expand network.** The HP NonStop™ Kernel operating system network that extends the concept of fault-tolerant operation to networks of geographically distributed HP NonStop S-series systems. If the network is properly designed, communication paths are constantly available even if there is a single line failure or component failure.

**Expand node.** A system in an [Expand network](#). See also [node](#).

**Expand node number.** A number in the range 0 through 254, sometimes referred to as the system number, that identifies a node in an Expand network. Each Expand node number must be unique within the network. See also [ServerNet node number](#).

**Expand-over-ServerNet line.** The single line associated with an Expand-over-ServerNet line-handler process. Note that an Expand-over-ServerNet line has the same name and logical device number as its Expand-over-ServerNet line-handler process. However, the line does not have the same states as the line-handler process.

**Expand-over-ServerNet line-handler process.** An Expand line-handler process that uses the NETNAM protocol to access the Network Access Method (NAM) interface provided by the ServerNet cluster monitor process, \$ZZSCL. The Expand-over-ServerNet line-handler process handles incoming and outgoing Expand messages. It also handles packets leaving the server and security-related messages going between systems within a ServerNet cluster. Each node in a ServerNet cluster must be configured with an Expand-over-ServerNet line-handler process for each other node in the ServerNet cluster.

**expansion service processor (ESP).** A service processor (SP) that is not a master service processor (MSP). ESPs occur in pairs in groups 02 through *nn* (not in group 01). See also [master service processor \(MSP\)](#).

**explicit DLL.** See [explicit dynamic-link library \(explicit DLL\)](#).

**explicit dynamic-link library (explicit DLL).** A dynamic-link library (DLL) that is named in the libList of a client or is a native-compiled loadfile associated with a client.

**export.** To offer a symbol definition for use by other loadfiles. A loadfile exports a symbol definition for use by other loadfiles that need a data item or function having that symbolic name.

**export digest.** A mathematical hash of the exported symbol names and locations in a library. Two libraries with the same export digest are interchangeable in that they both export the same symbols at the same locations; they are not necessarily semantically equivalent.

**extended data segment.** See [selectable segment](#).

**extended link format (ELF).** A standard binary file format common on UNIX systems. The ELF format is used for position-independent code (PIC) files.

**extensible input/output (XIO).** A redesign of the HP NonStop™ Kernel operating system's I/O subsystem to enable it to extend itself in general ways to meet future requirements.

**extent.** A contiguous area on disk for allocating one file.

**extent size.** The size in bytes of a contiguous area on disk for allocating one file.

**external entry point (XEP) table.** A table located in the last page of each TNS code segment that contains links for calls (unresolved external references) out of that segment.

**external fabric connection.** The low-level ServerNet connection between a node and one of the external ServerNet fabrics. Each node has an X and a Y connection to the external fabrics.

**external routing.** The routing of packets over the external ServerNet fabrics—that is, between systems (or nodes) in a ServerNet cluster. See also [internal routing](#).

**external ServerNet fabrics.** The fabrics that link systems in a ServerNet cluster. See also [internal ServerNet fabrics](#).

**external ServerNet X or Y fabric.** The X or Y fabric that links systems in a ServerNet cluster. See also [internal ServerNet X or Y fabric](#).

**external system area network manager process (SANMAN).** (1) A Guardian process with the name \$ZZSMN that provides management access to the external ServerNet X and Y fabrics. (2) A Windows NT process that configures and maintains ServerNet switches within a Windows NT cluster.

**fabric.** A complex set of interconnections through which there can be multiple and (to the user) unknown paths from point to point. The term “fabric” is used to refer to the X or Y portion of the ServerNet communications network; for example, the X fabric.

**factory-installed operating system.** The version of the operating system image that HP creates having a CONFTEXT configuration file, OSIMAGE file, and configuration



database that matches your order. Your system is shipped with this version installed in the system subvolume \$SYSTEM.SYS00.

**fan.** A component within an HP NonStop™ S-series system enclosure that circulates air into the enclosure to help maintain optimal temperature. Each NonStop S-series system enclosure contains two fans.

**far gateway.** A short code sequence that accomplishes the transition to privileged mode for legitimate calls to callable procedures that are located in a different direct jump area. Typically, SCr (system code, RISC) is the target area. See also [direct jump area](#), [far jump](#), and [gateway](#).

**far jump.** A sequence of RISC instructions that permits crossing the boundaries of the 256-megabyte direct jump areas in virtual memory. Such sequences are necessary, for example, when calling into system code from user code, because the two are located in different direct jump areas. The sequence ends with a JR (Jump via Register) RISC instruction.

**fastLoad.** An optimization that allows the loader to avoid reading symbols and binding symbolic references when loading a program or dynamic-link library in an environment equivalent to that of a previous load. See also [preset](#), [cached bindings](#), and [library import characterization \(LIC\)](#).

**Fast Ethernet ServerNet adapter (FESA).** A single-port ServerNet adapter that supports 100-megabit/second (Mbps) or 10-Mbps Ethernet data transfer rates on an HP NonStop™ S-series server. The 3863 FESA installs directly into an available I/O port.

**fault tolerance.** The ability of a computer system to continue processing despite the failure of any single software or hardware component within the system.

**feature-test macro.** In C and C++ programs, a symbol that, if defined in a program's source code, includes specific other symbols from a header within that program's source code and makes those symbols visible.

**feeder circuit.** The circuit conductors installed between the facility's main service entrance and the power distribution panels (PDPs) that supply the branch circuits.

**ferrule.** A cylindrical end terminal sometimes used on resistors, cartridge fuses, and other parts to permit quick insertion and removal from holders that have corresponding spring contacts.

**FESA.** See [Fast Ethernet ServerNet adapter \(FESA\)](#).

**Fiber Optic Extension (FOX).** Refers to two products, FOX II and ServerNet/FX, which allow you to create high-speed (up to 4 megabytes/second) networks of as many as 14 systems connected by dual fiber-optic cables.

**fiber-optic plug-in card (F-PIC).** A plug-in card (PIC) for the 6760 ServerNet device adapter (ServerNet/DA) that uses a fiber-optic interface to connect an HP NonStop™

S-series system to external disk drives and to some tape drives that contain a back-end board (BEB) that translates fiber-optic signals from the F-PIC into SCSI commands and information for the tape drive. See also [plug-in card \(PIC\)](#) and [SCSI plug-in card \(S-PIC\)](#).

**fiber optics.** A medium for data transmission that conveys light or images through very fine, flexible, glass or plastic fibers. Fiber-optic cables (light guides) are a direct replacement for conventional coaxial and wire pairs.

**fiber-optic ServerNet addressable controller (F-SAC).** A ServerNet addressable controller (SAC) that is contained within a fiber-optic plug-in card (F-PIC).

**fiber-optic ServerNet cable.** A ServerNet cable that uses fiber optics to transmit data. HP NonStop™ S-series servers support two types of fiber-optic ServerNet cables: [multimode fiber-optic \(MMF\) ServerNet cable](#) and [single-mode fiber-optic \(SMF\) ServerNet cable](#).

**field.** In a structured programming language, an addressable entry within a data structure. The term “field” is sometimes used to mean “[member](#).”

**field-programmable gate array (FPGA).** A programmable integrated circuit that can be customized to perform specific functions.

**field-replaceable unit (FRU).** A unit that can be replaced in the field only by qualified personnel trained by HP and cannot be replaced by customers. A unit is classified as a FRU because of safety hazards such as weight, size, sharp edges, or electrical potential; contractual agreements with suppliers; or national or international standards. See also [customer-replaceable unit \(CRU\)](#).

**FIFO.** A type of Open System Services (OSS) special file that is always read and written in a first-in, first-out manner.

**FIFO special file.** See [FIFO](#).

**file.** An object to which data can be written or from which data can be read. A file has attributes such as access permissions and a file type. In the Open System Services (OSS) environment, file types include regular file, character special file, block special file, FIFO, and directory. In the Guardian environment, file types include disk files, processes, and subdevices.

**file class.** The property of an Open System Services (OSS) file indicating access permissions for a process related to the owner, group, or other identification of the process. See also [file group class](#), [file other class](#), and [file owner class](#).

**file description.** See [open file description](#).

**file descriptor.** In the Open System Services (OSS) file system, the nonnegative integer that uniquely identifies a single open of a file to a running process. Each file descriptor is associated with an open file description that contains data about the file.

**file group class.** The property of an Open System Services (OSS) file indicating access permissions for a process related to the group ID of the process. A process is in the file group class of a file if both:

- The process is not a member of the file owner class for the file.
- The process has an effective group ID or supplementary group ID that is the same as the group ID associated with the file.

**file identifier.** In the Guardian environment, the portion of a filename following the subvolume name. In the Open System Services (OSS) environment, a file identifier is a portion of the internal information used to identify a file in the OSS file system (an inode number). The two identifiers are not comparable.

**file link count.** The total number of directory entries for an Open System Services (OSS) file within an HP NonStop™ node.

**file mode.** For an Open System Services (OSS) process, a field in the `stat` structure for a specific file that describes the type and characteristics of the file and contains the access permission bits for the file.

**file mode creation mask.** A mask associated with an Open System Services (OSS) process and used when the process creates a file. Bits set in this mask are cleared in the access permission bits for the file.

**filename.** In the Open System Services (OSS) environment, a component of a [pathname](#) containing any valid characters other than slash (/) or null. See also [file name](#).

**file name.** A string of characters that uniquely identifies a file.

In the PC environment, file names for disk files normally have at least two parts (the disk name and the file name); for example, B:MYFILE.

In the Guardian environment, disk file names include a node name, volume name, subvolume name, and file identifier; for example, \NODE.\$DISK.SUBVOL.MYFILE.

In the Open System Services (OSS) environment, a file is identified by a [pathname](#); for example, /usr/john/workfile. See also [filename](#).

**file other class.** The property of an Open System Services (OSS) file indicating access permissions for a process related to the user ID and group ID of the process. A process is in the file other class of a file if both:

- The process is not a member of the file owner class for the file.
- The process is not a member of the file group class for the file.

**file owner class.** The property of an Open System Services (OSS) file indicating access permissions for a process related to the user ID of the process. A process is in the file owner class of a file if the process has an effective user ID that is the same as the user ID (owner) associated with the file.

**file permission bits.** Information about an Open System Services (OSS) file that is used, along with other information, to determine whether a process or user has read, write, or execute/search permission to that file. The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of processes.

**file serial number.** A number that uniquely identifies a file within its file system.

**fileset.** In the Open System Services (OSS) environment, a set of files with a common mount point within the file hierarchy. A fileset can be part or all of a single virtual file system.

On an HP NonStop™ system, the Guardian file system for a node has a mount point and is a subset of the OSS virtual file system. The entire Guardian file system therefore could be viewed as a single fileset. However, each volume, and each process of subtype 30, within the Guardian file system is actually a separate fileset.

The term “file system” is often used interchangeably with “fileset” in UNIX publications.

**file system.** In the Open System Services (OSS) environment, a collection of files and file attributes. A file system provides the namespace for the file serial numbers that uniquely identify its files. Open System Services provides a file system (see also ISO/IEC IS 9945-1:1990 [ANSI/IEEE Std. 1003.1-1990], Clause 2.2.2.38); the Guardian application program interface (API) provides a file system; and OSS Network File System (NFS) provides a file system. (OSS NFS filenames and pathnames are governed by slightly different rules than OSS filenames and pathnames.) Within the OSS and OSS NFS file systems, filesets exist as manageable objects.

On an HP NonStop™ system, the Guardian file system for a node is a subset of the OSS virtual file system. Traditionally, the API for file access in the Guardian environment is referred to as the “Guardian file system.”

In some UNIX and NFS implementations, the term “file system” means the same thing as “fileset.” That is, a file system is a logical grouping of files that, except for the root of the file system, can be contained only by directories within the file system. See also [fileset](#).

**File Transfer, Access, and Management (FTAM).** The Open Systems Interconnection (OSI) standard developed by the International Organization for Standardization (ISO) for network file exchange and management services.

**file transfer protocol (FTP).** (1) The Internet-standard, high-level protocol for transferring files from one machine to another. The server side requires the client to supply a login identifier and password before it honors requests. FTP makes no assumptions about the file-naming structure of the source and destination systems, and it allows the file names of each system to be represented in the vernacular. (2) The application used to send complete files over Transmission Control Protocol/Internet Protocol (TCP/IP) services.

**filler panel.** A blank faceplate that is installed in place of a ServerNet adapter or plug-in card (PIC) to ensure proper ventilation.

**fingerprint.** A unique identifier calculated for a file and displayed in hexadecimal format.

**FIPS.** A Federal Information Processing Standard of the United States government.

**FIPS 151-1.** The Federal Information Processing Standard that specifies the requirements for conformance to an older draft of POSIX.1 (IEEE Std. 1003.1-1988) than the version adopted as ISO/IEC IS 9945-1:1990 and imposes some additional requirements.

**FIPS 151-2.** The Federal Information Processing Standard that specifies the requirements for conformance to POSIX.1 as ISO/IEC IS 9945-1:1990 and imposes some additional requirements.

**FIR.** See [FRU information record \(FIR\)](#).

**FIRINIT.** A diagnostic task used to update the communications line interface processor (CLIP) FRU information record (FIR) that is kept in the ServerNet wide area network (SWAN) concentrator CLIP flash memory.

**FIRMUP.** A diagnostic task used to update the copy of the Portable Silicon Operating System (pSOS) system product embedded kernel that is kept in the ServerNet wide area network (SWAN) concentrator communications line interface processor (CLIP) flash memory.

**firmware.** Code in memory that is necessary for the power-up initialization and communication with a host or device. The software for components of the ServerNet architecture (for example, an adapter) is called firmware. Some firmware for ServerNet components is downloaded when the system or component is loaded.

**fixed process configuration.** Using Subsystem Control Facility (SCF) to configure a generic process to always start in the first available processor (that is, to be fault tolerant).

**flag.** In a UNIX or Open System Services (OSS) command, a character sequence that begins with a hyphen and is processed as a unit.

**flash memory.** A type of memory that contains essential firmware and nonessential firmware.

**flash PROM.** A type of programmable read-only memory (PROM) that is electrically reprogrammable.

**flat segment.** A type of logical segment. Each flat segment has its own distinct address range within the process address space that never overlaps the range of any other allocated segments. Thus all allocated flat segments for a process are always available for use concurrently. See also [logical segment](#) and [selectable segment](#).

**foreground process.** An Open System Services (OSS) process that belongs to a foreground process group.

**foreground process group.** In the Open System Services (OSS) environment, a process group whose members have privileges for access to their controlling terminal that are denied to processes in background process groups of that terminal. Each session with a controlling terminal has only one foreground process group for that terminal. Contrast with [background process group](#).

**foreground process group ID.** In the Open System Services (OSS) environment, the process group ID of a foreground process group.

**four-lane link.** The four single-mode fiber-optic (SMF) ServerNet cables that connect the two HP NonStop™ Cluster Switches on the same external fabric (for example, X1 and X2) in a [split-star topology](#).

**FOX.** See [Fiber Optic Extension \(FOX\)](#).

**FOXMON.** See [FOX monitor process](#).

**FOX monitor process.** The Fiber Optic Extension (FOX) monitor process for the ServerNet/FX adapter subsystem. The process name is \$ZZFOX.

**FOX ring.** The fiber-optic cabling that connects the nodes in a Fiber Optic Extension (FOX) cluster. This term is also used to refer to the topology of a FOX network.

**FPGA.** See [field-programmable gate array \(FPGA\)](#).

**F-PIC.** See [fiber-optic plug-in card \(F-PIC\)](#).

**frame.** (1) An assembly of sheet-metal parts that is an integral part of an enclosure and might contain peripherals or a [chassis](#), depending on the type of enclosure. The frame enables the enclosures to be stacked and has provisions for routing and securing cables. The frame of an enclosure has dimensions that conform to an industry-standard 19-inch rack. (2) A unit of transmission in some data communications protocols, usually containing header, data, and checksum fields. (3) In NonStop S-series processors, a 4096-byte unit of physical memory; also called a physical page.

**frame base.** An assembly consisting of casters, leveling pads, and frame sheet metal that is an integral part of a base enclosure.

**free list.** In the Open System Services (OSS) file system, the list of available inodes that can be allocated to files.

**frequency.** The number of complete cycles/second of sinusoidal variation. For alternating-current (AC) power lines, the most common frequencies are 60 hertz and 50 hertz.

**FRU.** See [field-replaceable unit \(FRU\)](#).

**FRU information record (FIR).** A collection of information that every field-replaceable unit (FRU) carries with it, such as part number, revision, track ID, and media access control (MAC) address.



**F-SAC.** See [fiber-optic ServerNet addressable controller \(F-SAC\)](#).

**FTAM.** See [File Transfer, Access, and Management \(FTAM\)](#).

**FTP.** See [file transfer protocol \(FTP\)](#).

**gateway.** (1) A device used to convert the message protocol of one network to that of another. (2) A short code sequence that accomplishes the transition to privileged mode for legitimate calls to callable procedures. See also [far gateway](#).

**GB.** See [gigabyte \(GB\)](#).

**GCSC.** See [Global Customer Support Center \(GCSC\)](#).

**general-purpose register (GPR).** One of a small number of undedicated high-speed memory locations in a processor.

**generic process.** A process created and managed by the Kernel subsystem; also known as a system-managed process. A common characteristic of a generic process is [persistence](#).

**GESA.** See [Gigabit Ethernet ServerNet adapter \(GESA\)](#).

**Gigabit Ethernet ServerNet adapter (GESA).** A single-port ServerNet adapter that provides 1000 megabits/second (Mbps) data transfer rates between HP NonStop™ S-series systems and Ethernet LANs. A GESA can be directly installed in slots 51 through 54 of an I/O enclosure and slots 53 and 54 of a processor enclosure.

Two versions of the GESA are available:

- 3865 GESA-C (T523572): a single-port copper version compliant with the 1000 Base-T standard (802.3ab)
- 3865 GESA-F (T523572): a single-port fiber version compliant with the 1000 Base-SX standard (802..z)

**gigabyte (GB).** A unit of measurement equal to 1,073,741,824 bytes (1024 megabytes). See also [kilobyte \(KB\)](#), [megabyte \(MB\)](#), and [terabyte \(TB\)](#).

**Global Customer Support Center (GCSC).** A support organization that provides telephone and remote diagnostic support for HP customers. GCSCs are located all over the world. See also [Online Support Center \(OSC\)](#).

**global offset table (GOT).** A table of indirect addresses of data, including function descriptors, that might reside in a different loadfile. The GOT is an artifact of the native compiler.

**globalized.** The import-control characteristic of a loadfile that allows it to import symbols from any loadfile in the loadList of the program with which it is loaded. When those

loadfiles offer multiple definitions of the same symbol, those loadfiles are searched in loadList sequence and the first definition found takes precedence. See also [searchList](#).

**globally unique ID (GUID).** A unique, read-only number stored in nonvolatile memory (EEPROM) on a ServerNet II Switch at the time of manufacture. The GUID also appears on the bar code label. This number can be used programmatically to identify the switch.

**GOT.** See [global offset table \(GOT\)](#).

**GPR.** See [general-purpose register \(GPR\)](#).

**graphical user interface (GUI).** A user interface that offers point-and-click access to program functions.

**ground.** A conducting connection, whether intentional or accidental, between an electrical circuit and either the earth or some conducting body that serves in place of the earth, such as an underground metal water pipe, structural steel, or a ground rod driven into the earth. See also [earth ground](#).

**grounded.** Connected to earth or to some conducting body that serves in place of the earth.

**grounded conductor.** A system or circuit conductor that is intentionally grounded.

**ground fault.** Any undesired current path from a point of differing potential to ground.

**ground fault interrupter.** A device that interrupts the electric current to the load when a fault current to ground exceeds a predetermined value that is less than that required to operate the overcurrent protection device of the supply circuit.

**grounding conductor.** A conductor used to connect equipment of the grounded circuit of a wiring system to one or more earth-grounding electrodes. See also [earth-grounding electrode](#).

**group.** (1) The set of all objects accessible by a pair of service processors (SPs) located in the processor multifunction (PMF) customer-replaceable unit (CRU). In an HP NonStop™ S-series server, a system enclosure has exactly one group. (2) In the Open System Services (OSS) environment, a set of user IDs with the same group ID.

**group database.** A database on a node that contains the group name, group ID, and user names for each group using that node.



**group ID.** The nonnegative integer used to identify a group of users of an HP NonStop™ network node. Each user of a node is a member of at least one group. When the identity of a group is associated with an Open System Services (OSS) process, a group ID value is referred to as one of the following:

- Real group ID
- Effective group ID
- Supplementary group ID
- Saved-set group ID

**group list.** An Open System Services (OSS) process attribute that is used with the effective group ID of the process to determine the file access permissions for the process.

**GRT.** See [Guided Replacement Toolkit \(GRT\)](#).

**Guardian.** An environment available for interactive or programmatic use with the HP NonStop™ Kernel operating system. Processes that run in the Guardian environment usually use the Guardian system procedure calls as their application program interface; interactive users of the Guardian environment usually use the HP Tandem Advanced Command Language (TACL) or another HP product's command interpreter. Contrast with [Open System Services \(OSS\)](#).

**Guardian environment.** The Guardian application program interface (API), tools, and utilities.

**Guardian services.** An application program interface (API) to the HP NonStop™ Kernel operating system, plus the tools and utilities associated with that API. This term is synonymous with “Guardian environment.” See also [Guardian](#).

**Guardian user ID.** See [HP NonStop™ Kernel user ID](#).

**GUI.** See [graphical user interface \(GUI\)](#).

**GUID.** See [globally unique ID \(GUID\)](#).

**guided procedure.** A software tool that assists you in performing complex configuration or replacement tasks on an HP NonStop™ S-series server. Guided procedures are accessible from the Start menu on your system console. Examples include Replace SEB or MSEB, Configure ServerNet Node, and Replace IOMF.

**Guided Replacement Toolkit (GRT).** A software product that guides you through online replacement of the following customer-replaceable units (CRUs) on HP NonStop™ S-series systems: I/O multifunction (IOMF) CRUs, power supplies, processor multifunction (PMF) CRUs, and 6760 ServerNet device adapters. GRT is used only with older versions of TSM server software; if you are replacing a CRU in a system running TSM server version T7945AAX (shipped with G06.13) or later, use the appropriate [guided procedure](#).

**hard link.** In the Open System Services (OSS) file system, the relationship between two directory entries for the same file. A hard link acts as an additional pointer to a file. A hard link cannot be used to point to a file in another fileset. Contrast with [symbolic link](#).

**hard reset.** An action performed on an [HP NonStop™ Cluster Switch \(model 6770\)](#) and [HP NonStop™ ServerNet Switch \(model 6780\)](#) that reinitializes the router-2 ASIC within the switch, disrupting the routing of ServerNet messages through the switch for several minutes. When the hard reset is finished, the paths are restored automatically.

**harmonic.** The sinusoidal component of an alternating-current (AC) voltage that is a multiple of the waveform frequency.

**harmonic distortion.** Harmonics that change an alternating-current (AC) waveform from sinusoidal to complex.

**header.** An object that, when specified for inclusion in a program's source code, causes the program to behave as if the statement including the header were actually a specific set of other programming statements. A header contains coded information that provides details (such as data item length) about the data that the header precedes.

In an Open System Services (OSS) program, a header is the name of a file known to the run-time library used by a process. In a Guardian environment C language program, a header is the file identifier for a file known to the run-time library used by a process.

**hertz (Hz).** A unit of frequency. One hertz equals one cycle/second.

**high frequency.** A Federal Communications Commission (FCC) designation for a frequency in the range 30 through 300 megahertz, corresponding to a decametric wave in the range 10 through 100 meters.

**high PIN.** A [process identification number \(PIN\)](#) that is greater than 255. Contrast with [low PIN](#).

**hop count.** The number of routers that form a route between a ServerNet source and ServerNet destination. Hop count is used to determine the best route. If two alternate routes have the same time factor, the path with the lower hop count is the better route.

**host database.** An SQL database maintained for the host system and containing information about requests, software inputs, snapshots, targets, and profiles.

**host system.** (1) A computer system that supports very large databases and does batch processing, usually for an entire network of smaller systems. (2) The central site on which the Distributed Systems Management/Software Configuration Manager (DSM/SCM) is managed, the Archive is maintained, and configuration revisions are built. The host system is also a target system.

**HP NonStop™ Cluster Switch (model 6770).** An assembly that routes ServerNet messages across an external fabric of a ServerNet cluster. The cluster switch consists

of a ServerNet II Switch, an uninterruptible power supply (UPS), and AC transfer switch, and it can be packaged in a switch enclosure or in a 19-inch rack. The cluster switch is used with star, split-star, and tri-star topologies. See also [HP NonStop™ ServerNet Switch \(model 6780\)](#).

**HP NonStop™ ServerNet Switch (model 6780).** An assembly that routes ServerNet messages across an external fabric of a ServerNet cluster. The cluster switch consists of a ServerNet II Switch, an uninterruptible power supply (UPS), and AC transfer switch, and it can be packaged in a switch enclosure or in a 19-inch rack. The ServerNet switch is used with star, split-star, and tri-star topologies. See also [HP NonStop™ Cluster Switch \(model 6770\)](#).

**HP NonStop™ Kernel Open System Services (OSS).** The product name for the OSS environment. See also [Open System Services \(OSS\)](#).

**HP NonStop™ Kernel operating system.** The operating system for HP NonStop systems.

**HP NonStop™ Kernel user ID.** A [user ID](#) within an HP NonStop system. The Guardian environment normally uses the structured view of this user ID, which consists of either the *group-number*, *user-number* pair of values or the *group-name.user-name* pair of values. For example, the structured view of the super ID is (255, 255). The Open System Services (OSS) environment normally uses the scalar view of this user ID, also known as the [UID](#), which is the value  $(group-number * 256) + user-number$ . For example, the scalar view of the super ID is  $(255 * 256) + 255 = 65535$ .

**HP NonStop™ K-series servers.** The set of servers in the HP NonStop servers having product numbers beginning with the letter *K*. These servers run the HP NonStop Kernel operating system, but they do not implement the ServerNet architecture.

**HP NonStop™ S700 Server.** A special configuration of HP NonStop S-series server that is limited to one processor enclosure and a maximum of two I/O enclosures. A matched pair of any model of PMF CRU can be used in a NonStop S700 server.

**HP NonStop™ S7000 Server.** The first server in a product line of servers (HP NonStop S-series servers) that implement the ServerNet architecture and run the HP NonStop Kernel operating system.

**HP NonStop™ S7400 Server.** A model of HP NonStop S-series server that provides a midrange upgrade option for migrating from an HP NonStop K-series server or a NonStop S7000 server.

**HP NonStop™ S7600 Server.** A model of HP NonStop S-series server that implements the ServerNet architecture and runs the HP NonStop Kernel operating system. The NonStop S7600 PMF CRU is based on the NonStop S74000 PMF CRU, and the NonStop S7600 server supports all S-series hardware products that are compatible with the NonStop S74000 server.

**HP NonStop™ S70000 Server.** See [HP NonStop™ Sxx000 Server](#).

**HP NonStop™ S72000 Server.** See [HP NonStop™ Sxx000 Server](#).

**HP NonStop™ S74000 Server.** See [HP NonStop™ Sxx000 Server](#).

**HP NonStop™ S76000 Server.** See [HP NonStop™ Sxx000 Server](#).

**HP NonStop™ S86000 Server.** See [HP NonStop™ Sxx000 Server](#).

**HP NonStop™ Sxx000 Server.** Any server in a family of high-performance servers (HP NonStop S-series servers) that implement the ServerNet architecture and run the HP NonStop Kernel operating system. This family includes the NonStop S70000, S72000, S74000, S76000, and S86000 servers.

**HP NonStop™ ServerNet Cluster (ServerNet Cluster).** The product name for the collection of hardware and software components that constitute a [ServerNet cluster](#).

**HP NonStop™ S-series servers.** The set of servers in the HP NonStop servers having product numbers beginning with the letter S. These servers implement the ServerNet architecture and run the HP NonStop Kernel operating system.

**HP NonStop™ servers.** The entire line of HP NonStop servers, including NonStop K-series and NonStop S-series servers.

**HP NonStop™ System RISC Model D processor (NSR-D processor).** The model designation for the TNS/R processor used in the HP NonStop S7400 Server.

**HP NonStop™ System RISC Model E processor (NSR-E processor).** The model designation for the TNS/R processor used in the HP NonStop S7600 Server.

**HP NonStop™ System RISC Model G processor (NSR-G processor).** The model designation for the TNS/R processor used in the HP NonStop S70000 Server.

**HP NonStop™ System RISC Model T processor (NSR-T processor).** The model designation for the TNS/R processor used in the HP NonStop S72000 Server.

**HP NonStop™ System RISC Model V processor (NSR-V processor).** The model designation for the TNS/R processor used in the HP NonStop S74000 Server.

**HP NonStop™ System RISC Model W processor (NSR-W processor).** The model designation for the TNS/R processor used in the HP NonStop S7000 Server.

**HP NonStop™ System RISC Model X processor (NSR-X processor).** The model designation for the TNS/R processor used in the HP NonStop S76000 Server.

**HP NonStop™ System RISC Model Y processor (NSR-Y processor).** The model designation for the TNS/R processor used in the HP NonStop S86000 Server.

**HP NonStop™ TCP/IP.** The HP implementation of [Transmission Control Protocol/Internet Protocol \(TCP/IP\)](#) for the HP NonStop servers. See also [Parallel Library TCP/IP](#).

**HP NonStop™ TCP/IP process.** An HP product that supports the Transmission Control Protocol/Internet Protocol (TCP/IP) layers. TCP/IP processes are used together with the communications line interface processor (CLIP) pNA+ to provide the transport layer between wide area network (WAN) I/O processes and data link control (DLC) tasks, between ConMgr and the Simple Network Management Protocol (SNMP) task, between the WANBoot process and BOOTP tasks, and between an OSM or TSM process and a DIAG task.

**HP NonStop™ TCP/IP subsystem.** A subsystem that allows the use of HP NonStop TCP/IP to access an HP NonStop S-series host from Macintosh computers, personal computers, and UNIX workstations. Applications running on a NonStop S-series system or in an Expand network can transparently exchange data with NonStop TCP/IP devices.

**HP NonStop™ Transaction Management Facility (TMF).** HP software that provides transaction protection and database consistency in demanding online transaction processing (OLTP) and decision-support environments. It gives full protection to transactions that access distributed SQL and Enscribe databases, as well as recovery capabilities for transactions, online disk volumes, and entire databases.

**HP Open System Management (OSM) Interface.** Replacement for TSM as the system management tool of choice for NonStop S-series servers. OSM provides the same functionality as TSM while overcoming limitations of TSM. OSM is required for support of new functionality released in G06.21 and later.

**HP Tandem Advanced Command Language (TACL).** The user interface to the HP NonStop™ Kernel operating system. The TACL product is both a command interpreter and a command language. Users can write TACL programs that perform complex tasks or provide a consistent user interface across independently programmed applications.

**HP Tandem Failure Data System (TFDS).** A diagnostic tool that is a component of the HP NonStop™ Kernel operating system. The TFDS tool isolates software problems and provides automatic processor-failure data collection, diagnosis, and recovery services.

**HP Transaction Application Language (TAL).** A systems programming language with many features specific to stack-oriented TNS systems.

**hybrid shared run-time library (hybrid SRL).** A shared run-time library (SRL) that has been augmented by the addition of a dynamic section that exports the SRL's symbols in a form that can be used by position independent code (PIC) clients. A hybrid SRL looks like a dynamic-link library (DLL) to PIC clients (except it cannot be loaded at other addresses and cannot itself link to DLLs). The code and data in the SRL are no different in a hybrid SRL, and its semantics for non-PIC clients are unchanged.

**Hz.** See [hertz \(Hz\)](#).

**I18N.** See [internationalization](#).

**IBC.** See [in-band control \(IBC\)](#).

**ICMP.** See [Internet control message protocol \(ICMP\)](#)

**identifier.** A unique name, for example, TANDEM^FILES^TO^COPY, in the CONFTEXT file that refers to a text string (one or more file names as given in the CONFAUX file). When Distributed Systems Management/Software Configuration Manager (DSM/SCM) encounters an identifier, it substitutes the text string for the identifier.

**ideogram.** See [ideograph](#).

**ideograph.** A character or symbol representing a word or idea. Some writing systems, such as Japanese and Chinese, use thousands of ideographs. An ideograph is sometimes called an “ideogram.”

**IEC.** International Electrotechnical Committee. IEC is a professional organization that creates or adopts standards for computer hardware, environments, and physical interconnections.

**IEEE.** Institute of Electrical and Electronics Engineers. IEEE is a professional organization whose committees develop and propose computer standards that define the physical and data link protocols of entities such as communication networks.

**IEEE 802.3 protocol.** Institute of Electrical and Electronics Engineers (IEEE) standard defining the hardware layer and transport layer of (a variant of) Ethernet. The maximum segment length is 500 meters and the maximum total length is 2.5 kilometers. The maximum number of hosts is 1024. The maximum packet size is 1518 bytes.

**impedance.** The total opposition (that is, resistance and reactance) a circuit provides to the flow of alternating current at a given frequency.

**implementation-defined.** Not specified by a standard. A correct value or behavior that is implementation-defined can vary from system to system and therefore might represent a feature or facility that cannot be ported.

**implicit library.** A library supplied by HP that is available in the read-only and execute-only globally mapped address space shared by all processes without being specified to the linker or loader. See also [TNS system library](#) and [public library](#).

**implicit library import library (imp-imp).** See [import library](#).

**implied user library.** A method of binding TNS object files that have more than 16 code segments. Segments 16 through 31 are located in the user code (UC) space but are executed as if they were segments 0 through 15 of the user library (UL) code space. This method precludes the use of a user library. Binder now supports 32 segments of UC space concurrently with 32 segments of UL code space, so the implied user library method is not needed in new or changed TNS applications.



**import.** To refer to a symbol definition from another loadfile. A loadfile imports a symbol definition when it needs a data item or function having that symbolic name.

**import control.** The characteristic of a loadfile that determines from which other loadfiles it can import symbol definitions. The programmer sets a loadfile's import control at link time. That import control can be localized, globalized, or semiglobalized. A loadfile's import control governs the way the linker and loader construct that loadfile's searchList and affects the search only for symbols required by that loadfile.

**import library.** A file that represents a dynamic-link library (DLL) and can substitute for it as input to the linker. Import libraries facilitate linking on auxiliary platforms (that is, PCs) where it is inconvenient to store the actual DLLs.

**in-band control (IBC).** A symbol-based communications protocol for communicating management information across a ServerNet link without interfering with any application traffic in the network. In ServerNet II architecture, IBC traffic uses standard ServerNet packets. ServerNet I architecture uses the Illegal Symbol variation of IBC, which uses a subset of the available symbols to convey control information from one node to another. The symbol subset chosen is from the group of symbols that are not used for passing data; these symbols are usually considered illegal or unused.

**incident report.** A report sent by the OSM or TSM server software to the respective OSM or TSM Notification Director. If remote notification (dial-out) is configured, the Notification Director forwards incident reports to a service provider. There are three types of incident reports: problem incident reports, periodic incident reports, and software configuration incident reports.

**incremental discovery.** Discovery of an HP NonStop™ S-series server when the OSM or TSM client software has locally saved information but where there have been configuration changes on the server since that information was saved.

**indicator lights.** Two light-emitting diodes (LEDs) on a customer-replaceable unit (CRU) that indicate the status of the unit. The red or amber indicator light is lit when the unit is not working properly; during startup, this light can indicate that the unit is not yet functioning. The green indicator light is lit when the unit has proper power applied. See also [light-emitting diode \(LED\)](#).

**inductive reactance.** Resistance at a frequency that is caused by the inductance of a coil or circuit.

**initial discovery.** Discovery of an HP NonStop™ S-series server with which the OSM or TSM client software has had no prior contact and for which the client software has no locally saved information.

**initialization.** The process of defining a new Distributed Systems Management/Software Configuration Manager (DSM/SCM) target system, including giving it a name, setting up default values used when processing requests, and creating the first software revision (list of products) for the system.

**initial software revision.** The software revision on a target system when it is first brought into the Distributed Systems Management/Software Configuration Manager (DSM/SCM) environment. The DSM/SCM host database must be initialized with information about the initial software revision. The initial software revision is then used as a baseline upon which new software revisions are based.

**inode.** A data structure that stores the location of an Open System Services (OSS) file.

**inode number.** A unique identifier within the Open System Services (OSS) file system of an instance of an OSS file. The inode number identifies the instance within the file system catalogs.

**input/output process (IOP).** A running program (part of the HP NonStop™ Kernel operating system) that manages the I/O functions for one or more ServerNet addressable controllers (SACs) of the same type.

**input source.** The resource from which Subsystem Control Facility (SCF) accepts command input. SCF can accept input from a terminal or a disk file. The initial input source is determined by the form of the RUN command used to initiate SCF. At any time during an SCF session, the input source can be temporarily changed to execute a series of commands from a command file.

**inrush current.** The initial surge current demand of a load.

**Inspect region.** The region of a TNS object file that contains symbol tables for all blocks compiled with the SYMBOLS directive. The Inspect region is sometimes called the [symbols region](#).

**INSPSNAP.** The program that provides a process snapshot file for the Inspect subsystem. |

**installation subvolume (ISV).** A subvolume containing files that perform a specific function during the installation process, such as organizing documentation in a specific location, providing the components of the HP NonStop™ Kernel operating system image (OSIMAGE), and containing files that are used after the installation process.

**installer.** The person who installs the system equipment for a new system. This person also installs new equipment when additions are made to the system. This person can install software and perform system verification procedures as directed by the system planner, configuration planner, or support planner.

**instruction processing unit (IPU).** A processing unit that executes programs by fetching instructions from memory and executing them.

**insulated ground.** A grounding conductor with a dielectric (low-conductance) insulator around it to prevent inadvertent contact with metal conduits.

**intelligent SCSI processor (ISP).** The ServerNet addressable controller (SAC) that controls the small computer system interface (SCSI) bus.



**interactive mode.** A mode of operation that is characterized by having the same input and output device (a terminal or a process) for the session. If a terminal is used, a person enters a command and presses Return. If a process is used, the system interface waits for the process to send a request and treats the process in the same manner as a terminal. Contrast with [noninteractive mode](#).

**internal routing.** The routing of packets within an HP NonStop™ S-series server. See also [external routing](#).

**internal ServerNet fabrics.** The fabrics that link ServerNet devices within an HP NonStop™ S-series server. See also [external ServerNet fabrics](#).

**internal ServerNet X or Y fabric.** The X or Y fabric that links ServerNet devices within an HP NonStop™ S-series server. See also [fabric](#).

**internationalization.** The process of designing and coding software so that it can be adapted to meet the needs of different languages, cultures, and character sets, with the ability to handle various linguistic and cultural conventions. Internationalization methods enable the processing of character-based data independently of the underlying character encoding, allowing choice among character sets. Sometimes referred to as “I18N,” derived from the 18 letters between the initial “I” and the final “N” of the word “internationalization.” See also [character set](#).

**Internet address.** The 32-bit address assigned to hosts that want to participate in the Internet using Transmission Control Protocol/Internet Protocol (TCP/IP). Internet addresses are an abstraction of physical hardware addresses, just as the Internet is an abstraction of physical networks. As assigned to the interconnection of a host to a physical network, an Internet address consists of a network portion and a host portion. See also [IP address](#).

**Internet control message protocol (ICMP).** A maintenance protocol in the Transmission Control Protocol/Internet Protocol (TCP/IP) suite that is required in every TCP/IP implementation. The ICMP allows two nodes on an IP network to share IP status and error information. The ICMP is used by the ping utility to determine the readability of a remote system. See also [IP address](#) and [ping](#).

**Internet protocol (IP).** A data communications protocol that handles the routing of data through a network, which typically consists of many different subnetworks. IP is connectionless; it routes data from a source address to a destination address. See also [IP address](#).

**interoperability.** (1) Within an HP NonStop™ node, the ability to use the features or facilities of one environment from another. For example, the `gtac1` command in the Open System Services (OSS) environment allows an interactive user to start and use a Guardian tool in the Guardian environment. (2) Among systems from multiple vendors or with multiple versions of operating systems from the same vendor, the ability to exchange status, files, and other information. Product externals and end-user

publications for the NonStop range of servers often use the term “connectivity” in this context. See also [connectivity](#).

**interprocessor communications (IPC).** The exchange of messages between processors.

**intrinsic library.** See [Shared Millicode Library](#).

**instance.** A particular case of a class of items or objects. For example, a process is defined as one instance of the execution of a program; multiple processes might be executing the same program simultaneously. Also, instance data refers to global data of a program or library; each process has a its own instance of these data.

**instance data.** For each process using a dynamic-link library, a data segment area containing the global variables used by the library.

**I/O cabinet.** See [I/O enclosure](#).

**I/O controller.** The hardware logic that controls computer I/O operations for a particular set of devices, such as disks, tapes, terminals, or communications lines. See also [ServerNet addressable controller \(SAC\)](#).

**I/O enclosure.** An HP NonStop™ S-series system enclosure containing exactly one module, which includes ServerNet adapters, disk drives, components related to the ServerNet fabrics, and components related to electrical power and cooling for the enclosure. An I/O enclosure is identical to a processor enclosure, except that it contains I/O multifunction (IOMF) customer-replaceable units (CRUs) instead of processor multifunction (PMF) CRUs.

**IOMF CRU.** See [I/O multifunction \(IOMF\) CRU](#).

**IOMF 2 CRU.** See [I/O multifunction \(IOMF\) 2 CRU](#).

**I/O multifunction (IOMF) CRU.** (1) An HP NonStop™ S-series customer-replaceable unit (CRU) that connects an I/O enclosure to a processor enclosure through a ServerNet cable and supplies power to the components within the IOMF CRU as well as redundantly to the disk drives, SCSI terminators, and ServerNet adapters in that enclosure. The IOMF CRU contains a power supply, a service processor (SP), a ServerNet router, an Ethernet controller, an external ServerNet port, and three SCSI ServerNet addressable controllers (S-SACs) in a single unit. (2) A collective term for both IOMF CRUs and IOMF 2 CRUs when a distinction between the two types of CRUs is not required.

**I/O multifunction (IOMF) 2 CRU.** An HP NonStop™ S-series customer-replaceable unit (CRU) that connects an I/O enclosure to a processor enclosure through a ServerNet cable and supplies power to the components within the IOMF 2 CRU as well as redundantly to the disk drives, SCSI terminators, and ServerNet adapters in that enclosure. The IOMF 2 CRU contains a power supply, a service processor (SP), a ServerNet router 2, an Ethernet controller, three configurable ServerNet ports, and

three SCSI ServerNet addressable controllers (S-SACs) in a single unit. IOMF 2 CRUs are supported on G06.10 and later software release version updates (RVUs).

**IOP.** See [input/output process \(IOP\)](#).

**IP.** See [Internet protocol \(IP\)](#).

**IP address.** An address that uniquely identifies a specific host system within a network to the Internet protocol (IP). An IP address consists of two parts: a network address, which identifies the network, and a local address, which identifies the host within the network. IP routes data between source and destination IP addresses.

**iPAQ Desktop.** An Internet-based computing model and business PC used for Internet access, mainstream computing, and running the Windows 2000 Professional operating system. The iPAQ Desktop is designed to reduce hardware and software conflicts; it eliminates Industry Standard Architecture (ISA)/Peripheral Component Interconnect (PCI) slots and uses Universal Serial Bus (USB) ports with the Windows 2000 or Windows ME operating system.

**IPC.** See [interprocessor communications \(IPC\)](#).

**IPU.** See [instruction processing unit \(IPU\)](#).

**ISO.** International Organization for Standardization. ISO is an international body that drafts, discusses, proposes, and specifies standards for network protocols. ISO is best known for its seven-layer reference model that describes the conceptual organization of protocols.

ISO is sometimes called the “International Standards Organization”; although ISO is the official abbreviation, it does not correspond to the organization’s name in any language.

**ISO 646.** An ISO standard for representing characters in languages based on the Roman alphabet. Like ASCII, ISO 646 uses only 7 bits of each 8-bit byte to represent data. Contrast with [ISO 8859](#).

**ISO 8859.** A series of ISO standard 8-bit code sets used to represent languages based on many alphabets, including Roman, Greek, Cyrillic, Hebrew, Turkish, and Arabic. The ISO 8859 code sets are used in international applications that must be data transparent. ASCII is a subset of each of the ISO 8859 code sets.

**ISO 10646.** A universal coded character set that represents all characters and symbols from all commonly used scripts and languages.

**ISO/IEC-conforming POSIX.1 application.** An application that both:

- Uses only the facilities described in ISO/IEC IS 9945-1:1990 and approved conforming language bindings for any ISO or IEC standard.
- Is documented as using only those facilities and approved conforming language bindings.

**isolated ground.** A grounding conductor that directly connects the equipment ground through an isolated ground-type receptacle with the power system grounding point without any intermediate grounding points.

**isolation transformer.** A transformer containing electrostatic shields between the primary and secondary windings, with no direct electrical path between the primary and secondary windings.

**ISP.** See [intelligent SCSI processor \(ISP\)](#).

**ISV.** See [installation subvolume \(ISV\)](#).

**JDS box.** See [ServerNet extender module \(SEM\)](#).

**job control.** The Open System Services (OSS) features that allow processes to be stopped, continued, and moved from or to the background.

**KB.** See [kilobyte \(KB\)](#).

**Kernel subsystem.** In G-series release version updates (RVUs), the subsystem for configuration and management of the Subsystem Control Facility (SCF) subsystem managers that are generic processes, some system attributes, and the ServerNet X and Y fabrics.

**Kernel subsystem manager process.** The [generic process](#) that starts and manages other generic processes, some system attributes, and the ServerNet X and Y fabrics in G-series release version updates (RVUs). The \$ZZKRN Kernel subsystem manager process is started and managed by the \$ZPM persistence manager process.

**kilobyte (KB).** A unit of measurement equal to 1024 bytes. See also [gigabyte \(GB\)](#), [megabyte \(MB\)](#), and [terabyte \(TB\)](#).

**K-series servers.** See [HP NonStop™ K-series servers](#).

**L10N.** See [localization](#).

**labeled dump.** A token-by-token display-text representation of the Subsystem Programmatic Interface (SPI) command buffer or response buffer, as produced by the Subsystem Control Facility (SCF) commands DETAIL CMDBUFFER and DETAIL RSPBUFFER. The display text includes a labeled value for each token.

**LAN.** See [local area network \(LAN\)](#).

**LANMAN.** See [LAN manager \(LANMAN\) process.](#)

**LAN manager (LANMAN) process.** The process provided as part of the ServerNet local area network (LAN) Systems Access (SLSA) subsystem that starts and manages the SLSA subsystem objects and the LAN monitor (LANMON) process and assigns ownership of Ethernet adapters to the LANMON processes in the system. Subsystem Control Facility (SCF) commands are directed to the LANMON processes for configuring and managing the SLSA subsystem and the Ethernet adapters.

**LANMON.** See [LAN monitor \(LANMON\) process.](#)

**LAN monitor (LANMON) process.** The process provided as part of the ServerNet local area network (LAN) Systems Access (SLSA) subsystem that has ownership of the Ethernet adapters controlled by the SLSA subsystem.

**late binding.** At load time, binding a symbolic reference in a dynamic-link library (DLL) to a definition in a loadfile that appears on the program's loadList rather than the one found on the DLL's linker searchList. Late binding occurs in either of the following cases:

- The loader resolves a symbol that is unresolved by any loadfile on the linker searchList.
- The loader binds a symbol in a DLL to the first definition it finds on the program's loadList, and this is not the first definition that was encountered on the linker searchList.

For localized loadfiles, the linker and loader searchLists are the same, so late binding does not occur.

**layer number.** See [cluster switch layer number.](#)

**layered topology.** The network topology for ServerNet clusters using the [HP NonStop™ ServerNet Switch \(model 6780\)](#). The layered topology can scale by adding cluster switch layers or zones. The layered topology supports up to four layers and three zones. See also [star topology](#), [split-star topology](#), and [tri-star topology](#).

**LB.** See [logic board \(LB\)](#).

**LED.** See [light-emitting diode \(LED\)](#).

**legacy system.** An operating system that is not open but from which applications must be ported or users transferred.

**libList.** The list of libraries to be loaded along with a loadfile. When linking the loadfile, the linker constructs the libList from the names of libraries specified in the linker's command stream; it stores the libList within the loadfile.

**library.** A generic term for a collection of routines useful in many programs. An object code library can take the form of a linkfile to be physically included into client programs, it can be an OSS archive file containing several linkable modules, it can be a loadfile, or

it can be a system-managed collection of preloaded routines. Source-code libraries fall outside the scope of this glossary. See also [dynamic-link library \(DLL\)](#) and [shared run-time library \(SRL\)](#).

**library client.** A program or another library that uses routines or variables from that library.

**library file.** See [library](#).

**library import characterization (LIC).** A list of the export digests and relocation offsets of all the libraries used to resolve symbols in a loadfile. It allows the loader and operating system to determine when a file is being loaded in an environment equivalent to that found by the linker or to a previous load (see fastLoad). A LIC is generated and stored in the loadfile by the linker when a file is preset; it can be used in a subsequent load step to determine whether the loadfile's existing bindings can be reused. The operating system can also retain the bindings as modified when a loadfile is loaded (see cached bindings) and associate a LIC with those cached bindings, so that they can be reused when the same file is again loaded in an equivalent environment.

**LIC.** See [library import characterization \(LIC\)](#).

**LIF.** See [logical interface \(LIF\)](#).

**light-emitting diode (LED).** A semiconductor device that emits light from its surface. Indicator lights are composed of LEDs. See also [indicator lights](#).

**line.** The specific hardware path over which data is transmitted or received. A line can also have a process name associated with it that identifies an input/output process (IOP) or logical device associated with that specific hardware path.

**linear load.** Electrical loads for which the impedance is constant regardless of the voltage, so that if the voltage is sinusoidal, the current drawn is also sinusoidal.

**line-handler process.** See [Expand line-handler process](#) or [Expand-over-ServerNet line-handler process](#).

**line interface unit (LIU).** A dual-ported unit consisting of two parts: a communications line interface processor (CLIP) and a line interface module (LIM). An LIU can communicate with either the primary or the backup processor, providing fault tolerance. When it is a component of the communications subsystem, an LIU communicates with either processor through either of a pair of communications interface units (CIUs).

**link.** In the Open System Services (OSS) file system, a directory entry for a file.

**link count.** In the Open System Services (OSS) file system, the number of directory entries that refer to a particular file.

**linker.** (1) The process or server that invokes the message system to deliver a message to some other process or server. (2) A programming utility, which combines one or more



compilation units' linkfiles to create an executable loadfile for a native program or library.

**linkfile.** (1) For native C/C++ compilers in the Guardian environment, a command file for input to the `nld` or `ld` utility. (2) A file containing object code that is not yet ready to load and execute. Linkfiles are combined by means of a linker or binder to make an executable loadfile for a program or library. Compiling creates one linkfile per independent source module. Contrast with [loadfile](#).

**linking.** The operation of collecting, connecting, and relocating code and data blocks from one or more separately compiled object files to produce a target object file.

**link name.** In the Open System Services (OSS) environment, the [filename](#) associated with a specific file within a directory. The length of a filename, and therefore the length of a link name, depends on the file system.

**Linux.** Linus Torvald's version of the UNIX operating system. See also <http://www.linux.org>.

**listener.** The process or server that is notified by the message system that a message from some other process or server is being delivered.

**LIU.** See [line interface unit \(LIU\)](#).

**LMU.** See [logical memory unit \(LMU\)](#).

**load.** (1) To transfer the HP NonStop™ Kernel operating system image or a program from disk into a computer's memory so that the operating system or program can run. (2) To insert a tape into a tape drive, which prepares it for a tape operation (read or write).

**loadable library.** A loadfile that offers functions and data to other loadfiles. In this manual, dynamic-link libraries and hybrid shared run-time libraries are libraries. A library cannot normally be invoked externally; for example, by a RUN command. Instead, it is invoked by calls or data references from client loadfiles.

**loader.** A programming utility that transfers a program into memory so it can run. The mechanism that brings loadfiles into memory for execution, maps them into virtual address space, and resolves symbol references among them. Synonyms include run-time loader and run-time linker. The loader for TNS and for TNS/R native programs and libraries that are not position-independent code (PIC) is part of the operating system. For PIC loadfiles, a loader called RLD works with the operating system to load programs and libraries.

**loadfile.** An executable object code file that is ready for loading into memory and executing on the computer. Loadfiles are further classified as executable programs (containing a main routine at which to begin execution of that program) or executable libraries (supplying routines or variables to multiple programs or separately loaded libraries). A TNS code file might be both a loadfile and a linkfile; native code files are never both. Contrast with [linkfile](#).

**loadList.** A list of all the libraries that must be loaded for a given loadfile to execute. A loadfile's loadList includes all the libraries in the given loadfile's libList plus all the libraries in those loadfiles' libLists, and so forth. It does not include the implicit libraries. The loadList order is the sequence in which these loadfiles are to be loaded when they are not already loaded by a previous operation. The loadList of the program includes all the loadfiles present in the process, in the order they were loaded.

**local area network (LAN).** A network that is located in a small geographical area and whose communications technology provides a high-bandwidth, low-cost medium to which low-cost nodes can be connected. One or more LANs can be connected to the system such that the LAN users can access the system as if their workstations were connected directly to it. Contrast with [wide area network \(WAN\)](#).

**locale.** In localization, the definition of the subset of a user's environment that depends on language and cultural conventions.

**localization.** The process of adapting computer interfaces, data, and documentation to the culturally accepted way of presenting information in the culture. Sometimes referred to as "L10N," derived from the 10 letters between the initial "L" and the final "N" of the word "localization."

**localized.** The import-control characteristic of a loadfile that allows it to import symbols only from the loadfile itself followed by the libraries in its libList, libraries that those libraries re-export, and from these, any successions of re-exported libraries. See also [register-exact point](#) and [searchList](#).

**local mount.** In the Network File System (NFS), a mount that attaches the fileset associated with a server to the specified mount point within the local directory hierarchy. The local mount is visible within the NFS subsystem and makes the files associated with the server available through the path associated with the local mount point.

**local node.** See [local system](#).

**local operator.** The person who performs routine system operations, such as starting and stopping the system, loading and unloading tapes, and changing the air filter. The local operator is normally the operator of the asynchronous system console for the node. See also [operator](#).

**local processor.** A processor in the same node as the ServerNet cluster monitor process (SNETMON) that is reporting status about the processor.

**local system.** (1) An on-site system or a system that is geographically near to the user or operator. (2) From the perspective of a particular SNETMON, the system or node on which that SNETMON is running. (3) From the perspective of a system console operator, the system to which the operator is logged on. Contrast with [remote system](#).



**logical device name.** The name assigned to an I/O process during its configuration. Other processes use the logical device name when issuing Guardian procedure calls to the I/O process.

**logical device number.** A number that identifies a particular I/O device in the system. Logical device numbers are assigned to physical I/O devices.

**logical disk volume.** A hardware device or device pair that provides persistent, highly accessible storage for data on a medium that is either magnetic or optical.

**logical interface (LIF).** A process that allows an application or another process to communicate with data communications hardware.

**logical memory unit (LMU).** A group of four memory units. Memory on a processor and memory board (PMB) is divided into two LMUs. One LMU contains memory units in slots MS1 through MS4; the other LMU contains memory units in slots MS5 through MS8. An LMU must have memory units installed either in all of its slots or in none of its slots. See also [memory unit](#).

**logical segment.** A single data area consisting of one or more consecutive 128-kilobyte unitary segments that is dynamically allocated by a process. There are two types of logical segments: selectable segments and flat segments. See also [selectable segment](#) and [flat segment](#).

**logic board (LB).** (1) See [cluster switch logic board](#). (2) A printed wiring assembly (PWA) on which computer circuits (chips and wiring) are mounted. One type of logic board is a processor and memory board (PMB); another type is a multifunction I/O board (MFIOB).

**login.** The activity by which a user establishes a locally authenticated identity on a server. Each login has one login name.

**login name.** A user name associated with a session.

**logon sequence.** The process through which the HP NonStop™ S-series server to be managed is determined, the security constraints to interact with that server are met, and a connection with that server is established.

**low-level link.** A connection between the OSM or TSM client software running on a system console and the master service processors (MSPs) on an HP NonStop™ S-series server. When the HP NonStop Kernel operating system is not running, communication must take place over a low-level link. You can also communicate with a NonStop S-series server over a low-level link when the operating system is running. See also [service connection](#).

**low PIN.** A [process identification number \(PIN\)](#) that is in the range 0 through 254. Contrast with [high PIN](#).

**MAC address.** See [media access control \(MAC\) address](#).

**main bonding jumper.** The connection between the grounded circuit conductor and the equipment grounding conductor at the service.

**main memory.** Data storage, specifically the chips that store the programs and data currently in use by a processor. On HP NonStop™ S-series servers, main memory is stored on the processor and memory board (PMB) in the processor multifunction (PMF) customer-replaceable unit (CRU) and is cleared when the system is powered off.

**main service entrance.** The enclosure containing connection panels and switchgear, located at the point where the utility power lines enter the building.

**Maintenance Interface.** See [Archive and Database Maintenance Interface](#).

**management process.** A process through which an application issues commands to a subsystem. A management process can be part of a subsystem or it can be associated with more than one subsystem; in the latter case, the management process is logically part of each subsystem. Subsystem Control Point (SCP) is the management process for all subsystems controlled by Subsystem Control Facility (SCF).

**manager.** (1) For an HP NonStop™ system, the person responsible for day-to-day monitoring and maintenance tasks associated with a software subsystem on a NonStop node. (2) For a UNIX system, any person in Management and Information Services management for the site.

**man page.** A term sometimes used in UNIX documentation for the online or hard-copy version of a file that provides reference information. See [reference page](#).

**master service processor (MSP).** A service processor (SP) that provides the basic service processor functions as well as centralized system functions such as a console port, a modem port for remote support functions, and system-load control. The enclosure containing processors 0 and 1 (group 01) also contains a pair of MSPs. See also [expansion service processor \(ESP\)](#).

**MB.** See [megabyte \(MB\)](#).

**Measure.** A tool used for monitoring the performance of the HP NonStop™ servers. Measure can be used to check the performance of a ServerNet cluster.

**media access control (MAC) address.** A value in the Medium Access Control sublayer of the IEEE/ISO/ANSI local area network (LAN) architecture that uniquely identifies an individual station implementing a single point of physical attachment to a LAN.

**Media Interface Connector (MIC).** A type of head on a fiber-optic cable that has locking wings on the sides.

**megabyte (MB).** A unit of measurement equal to 1,048,576 bytes (1024 kilobytes). See also [gigabyte \(GB\)](#), [kilobyte \(KB\)](#), and [terabyte \(TB\)](#).

**member.** In a structured programming language, an addressable entry within a data structure. A member can be a simple field or a data structure.

**memory-exact point.** A potential breakpoint location within an accelerated object file at which the values in memory (but not necessarily the values in registers) are the same as they would be if the object file were running in TNS interpreted mode or on a TNS system. Most source statement boundaries are memory-exact points. Complex statements might contain several such points: at each function call, privileged instruction, and embedded assignment. Contrast with [register-exact point](#) and [nonexact point](#).

**memory manager.** A HP NonStop™ Kernel operating system process that implements the paging scheme for virtual memory. This process services requests generated by different interrupt handlers as well as by other system processes.

**memory page.** A unit of virtual storage. In TNS systems, a memory page contains 2048 bytes. In TNS/R systems, the page size is determined by the memory manager and can vary, depending on the processor type.

**memory slot.** One of eight slots for memory units on the processor and memory board (PMB). The slots are labeled MS1 through MS8.

**memory unit.** A unit consisting of a dual inline memory module (DIMM) or a single inline memory module (SIMM) that is installed in groups of four on the processor and memory board (PMB) of the processor multifunction (PMF) customer-replaceable unit (CRU). Memory units constitute the processor memory. The memory units in certain models of PMF CRU are not replaceable by customers or in the field.

**message monitor process (MSGMON).** A helper process for the ServerNet cluster monitor process (SNETMON) that runs in each processor on every node of a ServerNet cluster. MSGMON is started by the persistence manager process, \$ZPM. It performs duties for SNETMON in those instances where SNETMON needs an agent in each system processor. In addition, MSGMON monitors the connections within the processor and reports changes back to SNETMON when required.

**MFIOB.** See [multifunction I/O board \(MFIOB\)](#).

**MIC.** See [Media Interface Connector \(MIC\)](#).

**microcode.** Any machine code or data that can run in a microprocessor. HP produces two types of microcode for HP NonStop™ S-series systems: volatile and nonvolatile. Volatile microcode is loaded into the volatile random-access memory (RAM) of some types of printed wiring assemblies (PWAs) and is not retained in a host PWA when power to the PWA is interrupted. For nonvolatile microcode, see [firmware](#). See also [millicode](#).

**millicode.** RISC instructions that implement various TNS low-level functions such as exception handling, real-time translation routines, and library routines that implement the TNS instruction set. Millicode is functionally equivalent to TNS [microcode](#).

**mirrored disk or volume.** A pair of identical disk drives that are used together as a single logical volume. One drive is considered primary and the other is called the mirror. Each byte of data written to the primary drive is also written to the mirror drive; if the primary drive fails, the mirror drive can continue operations. See also [volume](#).

**MMF PIC.** See [multimode fiber-optic \(MMF\) plug-in card \(PIC\)](#).

**MMF ServerNet cable.** See [multimode fiber-optic \(MMF\) ServerNet cable](#).

**mode.** The set of attributes that specify the type and access permissions for a file. See also [file mode](#).

**modular ServerNet expansion board (MSEB).** A [ServerNet expansion board \(SEB\)](#) that uses plug-in cards (PICs) to provide a choice of connection media for routing ServerNet packets.

**module.** (1) A set of components sharing a common interconnection, such as a backplane. A module is a subset of a group, and it is usually contained in an enclosure. In an HP NonStop™ S-series server, there is exactly one module in a group. (2) A set of I/O devices or services that share a common protocol and can be controlled by a single module driver in the extensible I/O (XIO) subsystem.

**MON object type.** The Subsystem Control Facility (SCF) object type for the [Storage Management Foundation \(SMF\)](#) master process.

**mount.** To make a fileset accessible to the users of a node.

**mount point.** In the Open System Services (OSS) file system, a directory that contains a mounted fileset. The mounted fileset can be in a different file system.

**MRouter.** A field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC) that is part of the serial maintenance bus (SMB) architecture. The MRouter distributes the SMB throughout a group.

**MSEB.** See [modular ServerNet expansion board \(MSEB\)](#).

**MSEB CBB.** See [MSEB common base board \(CBB\)](#).

**MSEB common base board (CBB).** In modular ServerNet expansion boards (MSEBs), the printed wiring assembly (PWA) that plug-in cards (PICs) are installed on.

**MSEB port.** A connector on modular ServerNet expansion boards (MSEBs) used for ServerNet links. An MSEB has four fixed serial-copper ports and six [plug-in card \(PIC\)](#) slots that accept a variety of connection media. See also [SEB port](#).

**MSGMON.** See [message monitor process \(MSGMON\)](#).

**MSP.** See [master service processor \(MSP\)](#).

**multifunction I/O board (MFIOB).** A ServerNet adapter that contains ServerNet addressable controllers (SACs) for SCSI and Ethernet; a service processor; ServerNet links to the processor, to the two ServerNet adapter slots, and to one of the ServerNet expansion board (SEB) slots; and connections to the serial maintenance bus (SMB), which connects components within an enclosure to the service processor.

**multilane link.** A communication link between HP NonStop™ Cluster Switches that can consist of multiple ServerNet cables. Two-lane links and four-lane links are examples of multilane links.

**multimode fiber-optic (MMF) plug-in card (PIC).** A plug-in card (PIC) for the modular ServerNet expansion board (MSEB) and I/O multifunction (IOMF) 2 customer-replaceable unit (CRU) that supports the multimode fiber-optic (MMF) interface.

**multimode fiber-optic (MMF) ServerNet cable.** A fiber-optic cable that either allows more than one mode to propagate or supports propagation of more than one mode of a given wavelength. MMF ServerNet cable typically supports shorter transmission distances than [single-mode fiber-optic \(SMF\) ServerNet cable](#).

**multiplexed.** The action of separating data traffic from one line onto several distinct lines or of combining data traffic from several distinct lines onto one line.

**mutex.** See [mutual exclusion \(mutex\)](#).

**mutual exclusion (mutex).** An operating mode with interrupts disabled.

**NAM.** See [Network Access Method \(NAM\)](#).

**national-standards-body conforming POSIX.1 application.** An application that both:

- Uses only the facilities described in ISO/IEC IS 9945-1:1990 and approved standards of a specific member of the ISO or IEC (the national standards body).
- Documents use of only those facilities and approved standards and documents all options and dependencies on limits.

**native.** An adjective that can modify the following: object code, object file, process, procedure, and mode of process execution. Native object files contain native object code, which directly uses the MIPS or Itanium processor's instruction set and the corresponding conventions for register handling and procedure calls. Native processes are those created by executing native object files. Native procedures are units of native object code. Native mode execution is the state of the process when it is executing native procedures.

**native link editor.** See [nld utility](#).

**native mode.** See [TNS/R native mode](#).

**native system library.** Synonym for [implicit library](#).

**\$NCP.** The process name of the [network control process](#).

**NEC.** National Electrical Code.

**network.** Two or more computer systems (nodes) connected so that they can exchange information and share resources. See also [Expand network](#), [wide area network \(WAN\)](#), and [local area network \(LAN\)](#).

**Network Access Method (NAM).** The interface through which an Expand-over-ServerNet line-handler process communicates with the ServerNet cluster monitor process (SNETMON).

**network control process.** A process pair, named \$NCP, that runs in each system of an Expand network. \$NCP establishes and terminates system-to-system connections, maintains network-related system tables (including the network routing table, NRT), calculates the most efficient way to transmit data to other systems in the network, monitors and logs changes in the status of the network and its systems, informs \$NCPs at neighbor systems of changes in line or Expand line-handler process status, and aborts pending requests when all paths go down. See also [network routing table \(NRT\)](#).

**Network Information Service (NIS).** A distributed name service (formerly known as Yellow Pages) developed by Sun Microsystems. See also [Domain Name System \(DNS\)](#).

**network routing table (NRT).** A table that resides in each processor in each system in a network. The NRT associates each destination system with the logical device (LDEV) number of the best-path route Expand line-handler process to use to send messages to that system. See also [network control process](#).

**network topology.** The physical layout of components that define a system, a local area network (LAN), or a wide area network (WAN).

**neutral.** (1) The conductor used as the primary return for current during normal operation of electrical equipment. (2) The junction of the legs in a wye circuit. See also [wye](#).

**NIS.** See [Network Information Service \(NIS\)](#).

**nld utility.** (1) The utility that collects, links, and modifies code and data blocks from one or more object files to produce a target TNS/R native object file. The nld utility is similar to the Binder program used in the TNS development environment. (2) The native link editor invoked during system generation to build the TSYSCLR and TSYS DP2 files.

**NNA.** See [node-numbering agent \(NNA\)](#).

**NNA PIC.** See [node-numbering agent \(NNA\) plug-in card \(PIC\)](#).

**node.** (1) A uniquely identified computer [system](#) connected to one or more other computer systems in a network. See also [Expand node](#) and [ServerNet node](#). (2) An endpoint in a ServerNet fabric, such as a processor or ServerNet addressable controller (SAC).



**node number.** A number used to identify a member system in a network. The node number is usually unique for each system in the network. See also [node](#) and [ServerNet node number](#).

**node-numbering agent (NNA).** A field-programmable gate array (FPGA) in a single-mode fiber-optic (SMF) plug-in card (PIC) that translates the node number of each ServerNet packet entering or exiting the external ServerNet fabrics.

**node-numbering agent (NNA) plug-in card (PIC).** A plug-in card (PIC) for the modular ServerNet expansion board (MSEB) that supports the node-numbering agent (NNA) interface.

**node routing ID.** See [ServerNet node routing ID](#).

**noncanonical input mode.** For an Open System Services (OSS) process, a terminal input mode in which data is made available to the process when a timer expires or when a certain number of characters have been entered. Noncanonical data is not grouped into logical lines of input. This mode is sometimes called block mode or transparent mode. Contrast with [canonical input mode](#).

**nonclustered.** Lacking the quality of belonging to a cluster.

**nonconfigured object.** An object that comes into existence after Subsystem Control Facility (SCF) is running and that was created in response to activity outside the SCF environment. An SCF STATUS command can display the name of a nonconfigured object, but its state is UNKNOWN.

**nondedicated (public) LAN.** A local area network (LAN) connected to the Ethernet ports on an Ethernet 4 ServerNet adapter (E4SA), Fast Ethernet ServerNet adapter (FESA), or Gigabit Ethernet ServerNet adapter (GESA). Unlike a dedicated service LAN, a public LAN supports the connection of many types of servers and workstations. System consoles can be connected to a public LAN, but such system consoles cannot use all the OSM or TSM client applications. See also [dedicated service LAN](#).

**nonessential firmware.** Code that is used for support routines such as self-test diagnostics and that can be overwritten during flash programming without affecting the next power-up operation. Contrast with [essential firmware](#).

**nonexact point.** A code location within an accelerated object file that is between memory-exact points. The mapping between the TNS program counter and corresponding RISC instructions is only approximate at nonexact points, and interim changes to memory might have been completed out of order. Breakpoints cannot be applied at nonexact points. Contrast with [memory-exact point](#) and [register-exact point](#).

**noninteractive mode.** A mode of operation that usually involves a command file (an EDIT file that contains a series of commands). Contrast with [interactive mode](#).

**nonlinear load.** Electrical load for which the instantaneous current is not proportional to the instantaneous voltage. Consequently, the local impedance varies with the voltage.

**nonsensitive command.** A command that can be issued by any user or program that is allowed access to a subsystem—that is, a command on which the subsystem imposes no further security restrictions. For Subsystem Control Facility (SCF), nonsensitive commands are those that cannot change the state or configuration of objects; most of them are information commands. Contrast with [sensitive command](#).

**NonStop™ Cluster Switch.** See [HP NonStop™ Cluster Switch \(model 6770\)](#). |

**NonStop™ zone.** A branch of the power-distribution system that provides power directly to HP NonStop computer equipment.

**NonStop™ Kernel operating system.** See [HP NonStop™ Kernel operating system](#).

**NonStop™ ServerNet Cluster.** See [HP NonStop™ ServerNet Cluster \(ServerNet Cluster\)](#).

**NonStop™ ServerNet Switch.** See [HP NonStop™ ServerNet Switch \(model 6780\)](#). |

**NonStop™ TCP/IP.** See [HP NonStop™ TCP/IP](#).

**NonStop™ TCP/IP process.** See [HP NonStop™ TCP/IP process](#).

**NonStop™ TCP/IP subsystem.** See [HP NonStop™ TCP/IP subsystem](#).

**NonStop™ TCP/IPv6.** An HP product that adds IP version 6 (IPv6) functionality to the parallel library TCP/IP product. IPv6 is a TCP/IP protocol that extends the IP version 4 (IPv4) of 32 bits to 128 bits. NonStop TCP/IPv6 can be run in three modes: INET (only IPv4 and is a direct replacement for parallel library TCP/IP), INET 6 (only IPv6), and Dual (both IPv4 and IPv6 communications).

**normal mode.** Electromagnetic interference that occurs between current-carrying conductors (for example, line to neutral).

**notification.** Another name for an [incident report](#) created by the OSM and TSM Notification Director. When incident reports are dialed out to service providers, this process is also referred to as [remote notification](#). |

**NRT.** See [network routing table \(NRT\)](#).

**NSR-D processor.** See [HP NonStop™ System RISC Model D processor \(NSR-D processor\)](#).

**NSR-E processor.** See [HP NonStop™ System RISC Model E processor \(NSR-E processor\)](#).

**NSR-G processor.** See [HP NonStop™ System RISC Model G processor \(NSR-G processor\)](#).

**NSR-T processor.** See [HP NonStop™ System RISC Model T processor \(NSR-T processor\)](#).



**NSR-V processor.** See [HP NonStop™ System RISC Model V processor \(NSR-V processor\)](#).

**NSR-W processor.** See [HP NonStop™ System RISC Model W processor \(NSR-W processor\)](#).

**NSR-X processor.** See [HP NonStop™ System RISC Model X processor \(NSR-X processor\)](#).

**NSR-Y processor.** See [HP NonStop™ System RISC Model Y processor \(NSR-Y processor\)](#).

**null object type.** A placeholder object type for the Subsystem Control Facility (SCF) NAMES and VERSION commands, which do not require explicit specification of a particular object type.

**null string.** In C and C++ programs, a character string that begins with a null character. This term is synonymous with “empty string.”

**OBEY file.** See [command file](#).

**object.** One or more of the devices, lines, processes, and files in a subsystem; any entity subject to independent reference or control by one or more subsystems. In the Subsystem Control Facility (SCF), each object has an [object type](#) and an [object name](#).

**object-code library.** Synonym for [library](#).

**object code file.** A file containing compiled machine instructions for one or more routines. This file can be an executable loadfile for a program or library or a not-yet-executable linkfile for some program module. On other systems, an object code file is also known as a “binary” or as an “executable.”

**object name.** A unique name for a Subsystem Control Facility (SCF) object within a subsystem.

**object-name template.** A name that stands for more than one Subsystem Control Facility (SCF) object. Such a name includes one or more wild-card characters, such as \* (asterisk) and ? (question mark). See also [wild-card character](#).

**object type.** The category of Subsystem Control Facility (SCF) objects to which a specific SCF object belongs; for example, a specific disk has the object type DISK and a specific terminal may have the object type SU. Each subsystem has a set of object types for the objects it manages.

**obsolescent.** An indication that a feature or facility exists for compatibility with older versions or drafts of a standard. Obsolescent features or facilities should not be used, because they might be removed from future versions of a standard and therefore might not be portable.

**ODP.** See [Optical Disk Process \(ODP\)](#).

**offline.** (1) Used to describe tasks that are performed outside of the control of an application or computer system. (2) Used to describe tasks that require system resources to be shut down. Contrast with [online](#).

**offline change.** Any change that requires system resources to be shut down. Offline changes are usually performed during a planned outage. Contrast with [online change](#).

**offline configuration.** Configuration performed offline by SYSGENR. If necessary, you edit the CONFTEXT configuration file to create a new configuration and then run the Distributed Systems Management/Software Configuration Manager (DSM/SCM) (which in turn runs SYSGENR) to generate a system image for the new configuration.

**ohm.** The standard unit for measuring resistance.

**online.** Used to describe tasks that can be performed while the HP NonStop™ Kernel operating system and system utilities are operational. Contrast with [offline](#).

**online change.** Any change that can be performed while an application or its system resources are operational. In some situations, online changes might temporarily affect subsystem and application availability. For example, altering the characteristics of a communications line might temporarily affect applications that use the communications line. Contrast with [offline change](#).

**Online Support Center (OSC).** The group of support specialists within the HP [Global Customer Support Center \(GCSC\)](#) who respond to telephone calls regarding system problems and diagnose malfunctioning systems using remote diagnostic links.

**open file.** In the Open System Services (OSS) file system, a file with a file descriptor.

**open file description.** In the Open System Services (OSS) file system, a data structure within an HP NonStop™ node that contains information about the access of a process or of a group of processes to a file. An open file description records such attributes as the file offset, file status, and file access modes. An open file description is associated with only one open file but can be associated with one or more file descriptors.

**open migration.** In the Open System Services (OSS) file system, the set of events and outcomes that occur when an open file description is inherited by a child process in a different processor than its parent process. Contrast with [open propagation](#).

**open propagation.** In the Open System Services (OSS) file system, the set of events and outcomes that occur when an open file description is inherited by a child process in the same processor as its parent process. Contrast with [open migration](#).

**Open SCSI.** A subsystem that provides the hardware and software for a SCSI-2 open interface that runs on HP NonStop™ S-series servers and to which developers can attach [small computer system interface \(SCSI\)](#) devices.

**open system.** A system with interfaces that conform to international computing standards and therefore appear the same regardless of the system's manufacturer. For example, the Open System Services (OSS) environment on HP NonStop™ systems conforms to international standards such as ISO/IEC IS 9945-1:1990 (ANSI/IEEE Std. 1003.1-1990, also known as POSIX.1), national standards such as FIPS 151-2, and portions of industry specifications such as the X/Open Portability Guide Version 4 (XPG4).

**Open System Services (OSS).** An open system environment available for interactive or programmatic use with the HP NonStop™ Kernel operating system. Processes that run in the OSS environment usually use the OSS application program interface; interactive users of the OSS environment usually use the OSS shell for their command interpreter. Synonymous with "Open System Services (OSS) environment." Contrast with [Guardian](#).

**Open System Services (OSS) environment.** The HP NonStop™ Kernel Open System Services (OSS) application program interface (API), tools, and utilities.

**Open System Services (OSS) Monitor.** A Guardian utility that accepts commands affecting OSS objects through an interactive Guardian interface named the Subsystem Control Facility (SCF).

**Open System Services (OSS) signal.** A signal model defined in the POSIX.1 specification and available to TNS processes and TNS/R native processes in the OSS environment. OSS signals can be sent between processes.

**Open Systems Interconnection (OSI).** A seven-layer network architecture model defined by the International Organization for Standardization (ISO). The two lowest layers deal with the physical connections and their protocols. The five upper layers deal with network services, such as network file transfers and accessing remote databases.

**Open Systems Interconnection Layer 2.** The data-link control level of the Open Systems Interconnection (OSI) model, composed of asynchronous or minimal line control protocols, byte-oriented or character-oriented protocols, and bit-synchronous or bit-oriented protocols. Data link protocols can be defined in terms of method of access of data, link relationship of stations, error detection scheme, error recovery, message formatting, logical half-duplex or full-duplex operation, code, and machine transparency.

**operating system image.** See [OSIMAGE](#).

**operational environment.** The conditions under which your system performs. These include the devices and communications lines that are made active and the system and application processes that are started at system startup.

**operator.** (1) A symbol—such as an arithmetic or conditional operator—that performs a specific operation on operands. (2) In Network Control Language (NCL), a lexical element used for working on terms in expressions. There are five types of operators: parenthetical, arithmetic, Boolean, relational, and string. (3) For an HP NonStop™ system, the person or program responsible for day-to-day monitoring and maintenance

tasks associated with the HP NonStop Kernel operating system and the hardware of a NonStop node. The operator issues commands to subsystems; retrieves, examines, and responds to event messages; or does any combination of those things. See also [local operator](#). Contrast with [administrator](#). (4) For a UNIX system, any interactive user of that system.

**operator message.** A message, intended for an operator, that describes a significant event on an HP NonStop™ S-series system. An operator message is the displayed-text form of an Event Management Service (EMS) [event message](#).

**optical disk cartridge.** A container that protects an optical disk platter from damage and allows easy handling. Each cartridge contains two optical disk volumes.

**optical disk drive.** An optical storage library component that holds optical disk cartridges during read and write operations.

**Optical Disk Process (ODP).** The software I/O process that controls an [optical storage library \(OSL\)](#) and its optical disk volumes.

**optical disk volume.** One side of an [optical disk cartridge](#).

**optical storage library (OSL).** A storage device consisting of an optical disk cabinet, optical disk drives, multiple storage cells for optical disk cartridges, a robot that automatically loads the cartridges into and unloads them from the drives, and a cartridge access port (CAP) where an operator can load cartridges into or remove cartridges from the OSL.

**option.** In a UNIX or Open System Services (OSS) command, a [flag](#) and its parameters or a flag without parameters.

**ordinary library.** A dynamic-link library (DLL) or shared run-time library (SRL) that is not public. See [ordinary dynamic-link library \(ordinary DLL\)](#).

**ordinary dynamic-link library (ordinary DLL).** A dynamic-link library (DLL) or shared run-time library (SRL) that is not public; the code file is found at run time and can be provided by the user. Contrast with [public dynamic-link library \(public DLL\)](#).

**orphaned process group.** In the Open System Services (OSS) environment, a process group in which the parent of every member either is also a member of the process group or is a member of a different session.

**orphan file.** In the Open System Services (OSS) environment, a file with no corresponding inode in the PXINODE file.

**orphan inode.** In the Open System Services (OSS) environment, an inode that appears in the PXINODE file but has no links in the PXLINK file.

**OSC.** See [Online Support Center \(OSC\)](#).

**OSCONFIG file.** In G-series release version updates (RVUs), a configuration file built during system generation that contains only Software Problem Isolation and Fix Facility (SPIFF) and Software Identification (SWID) tool records. In D-series and earlier RVUs, the Configuration Utility Program (COUP) uses the \$SYSTEM.SYS<sub>nn</sub>.OSCONFIG file to store its configuration information.

**OSI.** See [Open Systems Interconnection \(OSI\)](#).

**OSIMAGE.** A file built during system generation that contains the complete image of the HP NonStop™ Kernel operating system that runs in each processor in the system.

**OSL.** See [optical storage library \(OSL\)](#).

**OSL object type.** The Subsystem Control Facility (SCF) object type for an [optical storage library \(OSL\)](#).

**OSM.** See [HP Open System Management \(OSM\) Interface](#).

**OSM Event Viewer.** OSM replacement for the TSM Event Viewer.

**OSM Low-Level Link.** OSM replacement for the TSM Low-Level Link.

**OSM Notification Director.** OSM replacement for the TSM Notification Director.

**OSM Service Connection.** OSM replacement for the TSM Service Application.

**OSS.** See [Open System Services \(OSS\)](#).

**OSS environment.** See [Open System Services \(OSS\) environment](#).

**OSS Monitor.** See [Open System Services \(OSS\) Monitor](#).

**OSS process ID.** In the Open System Services (OSS) environment, the unique identifier that identifies a process during the lifetime of the process and during the lifetime of the process group of that process. See also [PID](#).

**OSS signal.** See [Open System Services \(OSS\) signal](#).

**OSS user ID.** See [HP NonStop™ Kernel user ID](#).

**outage.** Time during which a computer system is not capable of doing useful work. Outages can be planned or unplanned. From the end user's perspective, an outage is any time an application being used is not available. See also [planned outage](#) and [unplanned outage](#).

**outage minutes.** A metric for measuring outages that translates percentages into minutes/year of downtime.

**output destination.** The resource to which Subsystem Control Facility (SCF) sends its responses to commands. SCF can direct output to a disk file, an application process, a

terminal, or a printer. The initial output destination is determined by the form of the RUN command used to initiate SCF. The output destination can be changed dynamically during an SCF session.

**owner.** (1) In the case of a disk file, the user or program that created the file, or a user or program to whom the creator has given the file with the File Utility Program (FUP) GIVE command. (2) In the case of a process, the user or program that created the process or, if the PROGID option was specified in the FUP SECURE command for the code file, the user or program that owns the code file. (3) In the case of a token or other definition, the subsystem that provided the definition. (4) In the case of a subsystem, the company or organization that provides the subsystem, or the eight-character string identifying that company.

**packet.** A block of information that contains fields for addressing, sequencing information, possible priority indicators, and a portion of a message or an entire message. See also [ServerNet packet](#).

**page.** See [memory page](#).

**Parallel Library TCP/IP.** An HP product that provides increased performance and scalability over conventional Transmission Control Protocol/Internet Protocol (TCP/IP). Parallel Library TCP/IP coexists with conventional TCP/IP on HP NonStop™ S-series systems and supports Ethernet 4 ServerNet adapters (E4SAs), Fast Ethernet ServerNet adapters (FESAs), Gigabit Ethernet ServerNet adapters (GESAs), and ServerNet wide area network (SWAN) concentrators. See also [HP NonStop™ TCP/IP](#).

**PARAM.** An HP Tandem Advanced Command Language (TACL) command and a Subsystem Control Facility (SCF) command you can use to create a parameter and give it a value. The TACL process stores the values of parameters assigned by the PARAM command and sends the values to applications that request parameter values.

**parent directory.** A particular directory in the hierarchy of directories within a file system. The parent directory for a directory contains an entry for that specific directory and is identified in that directory as the directory immediately above it in the hierarchy. The parent directory for a file contains an entry for that file.

**parent process.** The process that created a given process, or (if the creating process has stopped) a process that has inherited a given process. See also [child process](#).

**parent process ID.** In the Open System Services (OSS) environment, an attribute of a child process determined by the parent process. The parent process ID is the OSS process ID of the current parent process.

**passthrough terminator.** See [SCSI passthrough terminator](#).

**path.** The route between a processor and a subsystem. If a subsystem is configured for fault tolerance, it has a primary path (from the primary processor) and a backup path (from the backup processor).



**pathname.** In the Open System Services (OSS) file system and Network File System (NFS), the string of characters that uniquely identifies a file within its file system. A pathname can be either relative or absolute. See also ISO/IEC IS 9945-1:1990 (ANSI/IEEE Std. 1003.1-1990 or POSIX.1), Clause 2.2.2.57.

**pathname component.** See [filename](#).

**pathname resolution.** In the Open System Services (OSS) environment, the process of associating a single file with a specified pathname.

**pathname-variable limits.** Limits that can vary within the Open System Services (OSS) file hierarchy; that is, the limits on a pathname variable that can vary according to the directory in which pathname resolution begins.

**path prefix.** In the Open System Services (OSS) environment, a pathname, with an optional final slash (/) character, that refers to a directory.

**PDC.** See [phase-loss detector/contacter \(PDC\)](#).

**PDP.** See [power distribution panel \(PDP\)](#).

**PDU.** See [computer-room power center \(CRPC\)](#).

**peak load current.** The maximum instantaneous load over a designated interval of time.

**PEEK.** A utility program that reports statistics on resource use in a processor. PEEK is used to ensure proper allocation of memory and processes in a system after system load.

**peer fabric.** The fabric on which an operation is not taking place. The X and Y fabrics are peers. If an action is being performed on one fabric, the other fabric is the peer fabric.

**peer service processors.** A pair of service processors (X and Y) in a service processor (SP) domain. Peer service processors function similarly to a fault-tolerant process pair in an HP NonStop™ K-series system. See also [service processor \(SP\)](#).

**pending incident report.** An incident report that has never been delivered to your service provider, either because delivery to both the primary and backup dial-out points was unsuccessful or because the incident report was generated at an unattended site.

**pending signal.** A signal that has been generated for a process but has not been delivered. Pending signals are usually blocked signals.

**periodic incident report.** A type of incident report that is generated periodically to test the connection to the service provider and report the current system configuration. The default frequency is 20 days.

**peripheral enclosure.** An enclosure that contains components related to one or more peripherals. The 519x tape subsystem is an example of a peripheral enclosure.

Peripheral enclosures are not part of the set of system enclosures. Contrast with [system enclosure](#).

**Peripheral Utility Program (PUP).** A utility used in D-series and earlier release version updates (RVUs) to manage disks and other peripheral devices. In G-series RVUs, similar functions are performed by the Subsystem Control Facility (SCF).

**persistence.** For the Subsystem Control Facility (SCF), the capability of a generic process to restart automatically if it was stopped abnormally. You configure this capability by specifying a nonzero AUTORESTART value in an ADD command.

**persistence count.** The number of times the \$ZPM persistence manager process will restart a generic process that has been terminated abnormally. A generic process with an AUTORESTART value of 10 (the maximum) is said to have a persistence count of 10. See also [persistence](#).

**persistence manager process.** The \$ZPM process that is started and managed by the \$ZCNF configuration utility process and that starts generic processes in G-series release version updates (RVUs) and manages their persistence.

**persistent configuration.** A configuration that remains the same from one system load to another.

**persistent process.** A process that must always be either waiting, ready, or executing. Persistent processes are usually controlled by a monitor process that checks on the status of persistent processes and restarts them, if necessary.

**phase-loss detector/contacter (PDC).** Equipment used to detect the interruption (for 50 milliseconds or longer) or the complete loss of one or more phases of power to computer equipment. Upon detection of a phase dropout, the contactor shuts down all input phases to the system equipment, thereby allowing smooth system shutdown and recovery.

**physical interface (PIF).** The hardware components that connect a system node to a network.

**physical link interfaces.** Communications standards defined by standards organizations. The following physical link interfaces are supported for the ServerNet wide area network (SWAN) concentrator: RS-232, RS-442, RS-449, V.35, and X.21.

**Physical view.** One of several views of a server available in the view pane of the Management window of the OSM Service Connection, TSM Service Application, and OSM and TSM Low-Level Link. A Physical view of a server is a view of all the enclosures and is intended to represent the actual floor plan at the site. A Physical view of an enclosure is a visual representation of the physical placement of supported resources inside the enclosure. See also [Connection view](#).

**PIB.** See [power interface board \(PIB\)](#).



**PIC.** See [plug-in card \(PIC\)](#).

**PID.** In the Open System Services (OSS) environment, a synonym for [process ID](#). OSS process ID is the preferred term in HP NonStop™ S-series system publications.

In the Guardian environment, PID is sometimes used to mean either:

- A Guardian process identifier such as the [process ID](#)
- The *cpu, pin* value that is unique to a process within a node (see [HP NonStop™ Kernel user ID](#))

**PIF.** See [physical interface \(PIF\)](#).

**PIN.** See [process identification number \(PIN\)](#).

**ping.** A utility used to verify connections to one or more remote hosts. The ping utility uses the [Internet control message protocol \(ICMP\)](#) echo request and echo reply packets to determine whether a particular IP system on a network is functional. The ping utility is useful for diagnosing IP network or router failures.

**pipe.** In the Open System Services (OSS) environment, an unnamed FIFO, created programmatically by invoking the `pipe()` function or interactively with the shell pipe syntax character (`|`). A shell pipe redirects the standard output of one process to become the standard input of another process. A programmatic pipe is an interprocess communication mechanism.

**planned outage.** Time during which a computer system is not capable of doing useful work because of a planned interruption. A planned outage can be time when the system or user application is shut down to allow for servicing, upgrades, backup, or general maintenance.

**planner.** The Distributed Systems Management/Software Configuration Manager (DSM/SCM) user who is responsible for planning and managing new software revisions. The planner uses the DSM/SCM Planner Interface to carry out these functions.

**Planner Interface.** A graphical user interface (GUI) to the Distributed Systems Management/Software Configuration Manager (DSM/SCM) that runs on the host system. It provides an interface to all the host DSM/SCM planner functions.

**plug-in card (PIC).** A replaceable component that provides a unique function when installed in a customer-replaceable unit (CRU) or field-replaceable unit (FRU). PICs for modular ServerNet expansion boards (MSEBs) and I/O multifunction (IOMF) 2 CRUs provide a choice of connection media for attaching ServerNet cables.

**PM.** See [product module \(PM\)](#).

**PMB.** See [processor and memory board \(PMB\)](#).

**PMCU.** See [power monitor and control unit \(PMCU\)](#).

**PMF CRU.** See [processor multifunction \(PMF\) CRU](#).

**PMF 2 CRU.** See [processor multifunction \(PMF\) 2 CRU](#).

**pNA+.** The support for the Transmission Control Protocol/Internet Protocol (TCP/IP) layers and the Ethernet interface provided by Integrated Systems Inc. as part of the Portable Silicon Operating System (pSOS) system product. pNA+ is provided as part of the wide area network (WAN) architecture in each ServerNet wide area network (SWAN) concentrator communications line interface processor (CLIP).

**Point-to-Point Protocol (PPP).** A data communications protocol that provides a standard method of encapsulating Transmission Control Protocol/Internet Protocol (TCP/IP) information over point-to-point links. OSM and TSM use PPP to provide TCP/IP communication over a dial-up connection.

**POOL object type.** The Subsystem Control Facility (SCF) object type for [Storage Management Foundation \(SMF\)](#) storage pools.

**port.** (1) A data channel that connects to other devices or computers. (2) A connector to which a cable can be attached. The system transmits and receives data or requests through ports on ServerNet adapters and processor multifunction (PMF) customer-replaceable units (CRUs). A port is also called a connector. (3) The entrance or physical access point (such as a connector) to a computer, multiplexer, device, or network where signals are supplied, extracted, or observed.

**portable application.** An application that can execute on a wide range of hardware systems from multiple manufacturers. A portable application is a program that can be moved with little or no change in its source code from another manufacturer's system to an HP NonStop™ system.

**portable filename character set.** The set of characters that includes the Roman uppercase and lowercase letters, the Arabic numerals, the period, the underscore, and the hyphen. The hyphen cannot be the first character of a portable filename.

**portable pathname character set.** The set of characters that includes the Roman uppercase and lowercase letters, the Arabic numerals, the period, the underscore, the slash (/), and the hyphen. The hyphen cannot be the first character of a portable pathname.

**Portable Silicon Operating System (pSOS) system product.** A product of Integrated Systems Inc. that provides support for industry-standard communications protocols based on the UNIX operating system. It is used as a compact multitasking kernel operating system for PowerPCs and similar systems.

**position ID.** A character that indicates the position an HP NonStop™ Cluster Switch occupies in a network topology. The position ID is a component of the two-character cluster switch name. The cluster switch name includes an external fabric ID (X or Y) as

the first character and a position ID as the second character. For example, the cluster switch name X3 indicates that the cluster switch serves the external ServerNet X fabric and occupies position 3 in the topology. Supported values for position IDs are 1 through 9 or A through Z. Currently supported topologies (star, split-star, and tri-star) use position IDs 1, 2, and 3.

**position-independent code (PIC).** Executable program or library code that is designed to be loaded and executed at any virtual memory address, without any modification. Addresses that can be modified by the loader do not appear in PIC code, only in data that can be modified by the loader. See also [dynamic-link library \(DLL\)](#).

**POSIX.** The Portable Operating System Interface, as defined by the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI). Each POSIX interface is separately defined in a numbered ANSI/IEEE standard or draft standard. The application program interface (API), known as POSIX.1, has become ISO/IEC IS 9945-1:1990.

**power distribution panel (PDP).** A group of panel assemblies that composes a single panel that includes buses and overcurrent protection devices (with or without switches). A PDP is used for the control of power circuits.

**power distribution unit (PDU).** See [computer-room power center \(CRPC\)](#).

**power domain.** A set of customer-replaceable units (CRUs) and field-replaceable units (FRUs) that share a set of power rails. For Telco Central Office (CO) systems, the power domain is the entire system.

**power factor.** The ratio of real power to apparent power (that is, kilowatts/kilovoltamperes). The power factor for a sinusoidal load is determined by the position of the applied voltage waveform with respect to the current drawn by the load. When voltage and current are in phase with each other, the power factor is unity and the power for the load is equal to the product of the applied voltage and load current ( $P=EI$ ). When the current waveform lags after the voltage waveform, the load is inductive. Conversely, when the current waveform leads the voltage waveform, the load is capacitive. In either case, the power for the load is equal to the product of the applied voltage, load current, and the angular displacement between the voltage and current waveforms ( $P=EI\cos\phi$ ). Nonlinear (nonsinusoidal) loads also have a power factor; however, the power factor for a nonsinusoidal load reflects harmonic content and not angular displacement.

**power factor correction.** The addition of a reactive component to offset the angular displacement of a sinusoidal load. Traditionally, the normal power factor for a facility is inductive, so the normal correction involves the addition of capacitors to offset the lagging power factor. The capacitors offset part or all of the inductive reactance, making the total circuit more nearly in phase with the applied voltage. The power factor for nonlinear (nonsinusoidal) loads cannot be corrected through the addition of simple reactive components. Harmonic filters are required to correct the power factor of nonlinear loads.

**power interface board (PIB).** In system enclosures with power shelves, a board mounted on the bulkhead located behind the power supplies in the power shelf. The PIB provides electrical connection between the power supplies and DC power cables.

**power monitor and control unit (PMCU).** A field-replaceable unit (FRU) that connects the batteries to the DC power distribution bus in an HP NonStop™ S-series enclosure and provides a means of disconnecting the batteries for powering off the system. The PMCU also provides a means for the service processor (SP) to diagnose the condition of the batteries, fans, and power supplies; to regulate the voltage supplied to the fans; and to provide the interface to the group ID switches and service light-emitting diodes (LEDs). A group contains two PMCUs, one for each of the two DC power distribution buses.

**power shelf.** In HP NonStop™ S7400, S7600, and Sxx000 processor enclosures and I/O enclosures containing I/O multifunction (IOMF) 2 customer-replaceable units (CRUs), an assembly residing below the chassis consisting of power supplies and supporting circuitry that provides DC power to the enclosure.

**power supply.** (1) In system enclosures without power shelves, the component on the processor multifunction (PMF) customer-replaceable unit (CRU) or the I/O multifunction (IOMF) CRU that converts standard AC line voltage into the DC voltages needed by the group components in the enclosure. (2) In system enclosures with power shelves, the component located in the power shelf that converts standard AC line voltage into DC voltage and delivers it to the PMF CRUs or IOMF CRUs in that enclosure, which in turn supply the DC voltages needed by the group components in the enclosure.

**PPP.** See [Point-to-Point Protocol \(PPP\)](#).

**preemption.** A form of late binding in which a symbolic reference to a symbol defined in the same dynamic-link library is instead bound to a definition in another loadfile.

**preferences file.** A file that contains configuration information for the graphical user interface (GUI) portion of the OSM and TSM client software. The preferences file is used by the OSM and TSM client software at system startup.

**preferred path.** See [primary path](#).

**preprocessing commands.** Commands specifying unique run-time parameters that can override your default system parameters. These commands can assign process file names, select backup media formats, and define utility options during system configuration.

**preset.** A linker operation that sets the correct values (addresses) of imported symbols according to the environment seen by the linker. If the loader encounters the same environment at load time, it avoids adjusting these values, which reduces loading overhead (see fastLoad). If not, the loader resets these values to match the load-time environment.

**primary path.** A path enabled as the preferred path. When a primary path is disabled, an [alternate path](#) becomes the primary path.

**primary processor.** The processor that is designated as “owning” the ServerNet addressable controller (SAC) connected to separate processors running the HP NonStop™ Kernel operating system. The primary processor is the processor that has direct control over the SAC. Contrast with [backup processor](#).

**private dynamic-link library (private DLL).** See [ordinary dynamic-link library \(ordinary DLL\)](#).

**problem incident report.** A type of incident report that reports a problem in the server. A problem incident report is generated when changes occur on the server that could directly affect the availability of system resources.

**procedure entry-point (PEP) table.** A table in a TNS object file that contains the entry point addresses for each procedure and is located in the first page of each code segment.

**process.** (1) A program that has been submitted to the operating system for execution, or a program that is currently running in the computer. (2) An address space, a single thread of control that executes within that address space, and the system resources required by that thread of control.

**process group.** In the Open System Services (OSS) environment, a set of processes that can signal associated processes. Each process in a node is a member of a process group; the process group has a process group ID. A new process becomes a member of the process group of its creator.

**process group ID.** In the Open System Services (OSS) environment, the unique identifier representing a process group during its lifetime.

**process group leader.** In the Open System Services (OSS) environment, the process that has the process group ID of its process group as its OSS process ID.

**process group lifetime.** In the Open System Services (OSS) environment, the period that begins when a process group is created and ends when the lifetime of the last remaining process of the group ends.

**process ID.** In the Guardian environment, the content of a four-integer array that uniquely identifies a process during the lifetime of the process. See also [PID](#).

**process identification number (PIN).** A number that uniquely identifies a process running in a processor. The same number can exist in other processors in the same system. See also [process ID](#).

**process image file.** On a UNIX system, an executable object file. In some Guardian product externals and end-user publications, an executable object file is referred to as a “program file.” See also [object code file](#).

**process lifetime.** The period that begins when an Open System Services (OSS) process is created and ends when its OSS process ID is returned to the system for reuse.

**PROCESS object type.** In a subsystem, the object type for the subsystem manager process itself or any generic process.

**processor.** (1) A functional unit of a computer that reads program instructions, moves data between processor memory and the input/output controllers, and performs arithmetic operations. A processor is sometimes referred to as a [central processing unit \(CPU\)](#), but the HP NonStop™ servers have multiple cooperating processors rather than a single CPU. (2) One or more computer chips, typically mounted on a logic board, that are designed to perform data processing or to manage a particular aspect of computer operations.

**processor and memory board (PMB).** A logic board that has lockstepped microprocessors, the main memory system, and the ServerNet memory interface (SMI) application-specific integrated circuits (ASICs) to act as an interface between the microprocessors and memory and the ServerNet fabrics. This board is part of the processor multifunction (PMF) customer-replaceable unit (CRU).

**processor cache.** A small, fast memory holding recently accessed data in order to speed up subsequent access to the same data. Cache memory is built from faster memory chips than main memory, and it is most often used with process or main memory but also used in network data transfer (to maintain a local copy of data), and so forth.

**processor dump.** A copy of the memory of a processor. A dump can be to disk or to tape. See also [ServerNet dump](#) and [tape dump](#).

**processor enclosure.** An HP NonStop™ S-series system enclosure containing exactly one group, which includes processors, ServerNet adapters, disk drives, components related to the ServerNet fabrics, and components related to electrical power and cooling for the enclosure.

**processor multifunction (PMF) CRU.** (1) An HP NonStop™ S-series customer-replaceable unit (CRU) that contains a power supply, service processor (SP), ServerNet router 1, Ethernet controller, three ServerNet addressable controllers (SACs), and a processor and memory system in a single unit. The PMF CRU consists of three subassemblies: the processor and memory board (PMB), the multifunction I/O board (MFIOB), and the power supply subassembly. (2) A collective term for both PMF CRUs and PMF 2 CRUs when a distinction between the two types of CRUs is not required.

**processor multifunction (PMF) 2 CRU.** An HP NonStop™ S-series customer-replaceable unit (CRU) that contains a power supply, service processor (SP), ServerNet router 2, Ethernet controller, three ServerNet addressable controllers (SACs), and a processor and memory system in a single unit. The PMF 2 CRU consists of three subassemblies: the processor and memory board (PMB), the multifunction I/O board (MFIOB), and the power supply subassembly.



**product module (PM).** The part of the Subsystem Control Facility (SCF) subsystem that is responsible for subsystem-specific command processing.

**profile.** Default values used by the Distributed Systems Management/Software Configuration Manager (DSM/SCM) when processing requests. There are three types of profiles: the Configuration Manager profile, the system profile, and the target profile.

**PROFILE object type.** The Subsystem Control Facility (SCF) object type for the storage subsystem configuration profile.

**program.** See [program file](#).

**program file.** An executable object code file containing a program's main routine plus related routines statically linked together and combined into the same object file. Other routines shared with other programs might be located in separately loaded libraries. A program file can be named on a RUN command; other code files cannot. See also [object code file](#).

**pSOS system product.** See [Portable Silicon Operating System \(pSOS\) system product](#).

**public dynamic-link library (public DLL).** Optional native-mode executable code modules available to all native user processes. A public library that is specified in the public library registry, supplied by HP or, optionally, a user. Contrast with [ordinary dynamic-link library \(ordinary DLL\)](#).

**public LAN.** See [nondedicated \(public\) LAN](#).

**public library.** A dynamic-link library (DLL) or shared run-time library (SRL) that is known to the operating system, available for execution by any process or user, and is not an implicit library.

**public shared run-time library (public SRL).** A TNS/R library supplied by HP.

**PUP.** See [Peripheral Utility Program \(PUP\)](#).

**quad-integrated communications controller (QUICC).** The Motorola MC68360 chip. For HP NonStop™ S-series servers, the QUICC is used as the service processor (SP) and is the main part of the ServerNet wide area network (SWAN) concentrator communications line interface processor (CLIP).

**quality power.** The attributes and configuration of the power-distribution systems installed within a facility that best serve the power needs of that facility's electrical equipment (for example, computer systems, air conditioning, and so on), providing the minimum possible disruption to equipment operation.

**QUICC.** See [quad-integrated communications controller \(QUICC\)](#).

**R1.** See [ServerNet router 1](#).

**R2.** See [ServerNet router 2](#).

**raceway.** An enclosed channel used to hold wires, cables, or busbars. Most raceways have removable tops to facilitate the installation or removal of their contents.

**rack.** A structure that houses a chassis, power shelf, and other system components. The HP NonStop™ S-series server is designed to be mounted in an industry-standard 19-inch rack or a NonStop S-series frame. See also [frame](#).

**radio frequency interference (RFI).** Forms of conducted or radiated interference that might appear in a facility as either normal or common-mode signals. The frequency of the interference can range from the kilohertz to gigahertz range. However, the most troublesome interference signals are usually found in the kilohertz to low megahertz range. At present, the terms radio frequency interference and [electromagnetic interference \(EMI\)](#) are usually used interchangeably.

**range of servers.** See [HP NonStop™ servers](#).

**read-only file system.** A file system with implementation-defined characteristics that restrict changes to the files within that file system.

**read/write head.** An electromagnet that can pick up (read) electronic pulses and record (write) electronic pulses on a magnetic disk or tape. The electronic pulses are interpreted by the processor as binary data. See also [disk drive](#) and [tape drive](#).

**real group ID.** An attribute of an Open System Services (OSS) process. When an OSS process is created, the real group ID identifies the group of the user or parent process that created the process. The real group ID can be changed after process creation.

**real user ID.** An attribute of an Open System Services (OSS) process. When an OSS process is created, the real user ID identifies the user or parent process that created the process. The real user ID can be changed after process creation.

**\$RECEIVE.** The name of a file through which a process receives and optionally replies to messages from other processes.

**reconfiguration.** The act of changing the hardware or software configuration of a running system. Examples include installing a new software release version update (RVU), adding hardware peripherals, and restructuring a database. Reconfiguring a system might or might not require a planned outage.

**reduced instruction-set computing (RISC).** A processor architecture based on a relatively small and simple instruction set, a large number of general-purpose registers, and an optimized instruction pipeline that supports high-performance instruction execution. Contrast with [complex instruction-set computing \(CISC\)](#).

**re-exported library.** A library whose symbols are made available by another dynamic-link library (DLL) to any localized client of that DLL. Re-export is an attribute of the DLL's libList entry for that library. This attribute is specified by the DLL's programmer and



recorded by the linker as a DLL is built. It affects only localized clients of the DLL. This feature allows a symbol to be moved from one DLL to another without relinking clients of the original DLL.

Re-exporting is transitive; that is, if A re-exports B and B re-exports C, then A re-exports C. Re-exported libraries can re-export other libraries to form a succession of re-exported-libraries of arbitrary length.

**reference page.** In Open System Services (OSS) and Distributed Computing Environment (DCE), the online or hard-copy version of a file that provides reference information for a software facility. Some UNIX product externals and end-user publications use the term “man page” instead, referring either to the online delivery mechanism used to display the file (usually the shell `man` command) or to the nature of the file as part of a publication.

**register-exact point.** A synchronization location within an accelerated object file at which both of these statements are true:

- All live TNS registers plus all values in memory are the same as they would be if the object file were running in TNS mode or TNS interpreted mode or on a TNS system.
- All accelerator code optimizations are ended.

Register-exact points are a small subset of all memory-exact points. Procedure entry and exit locations and call-return sites are usually register-exact points. All places where the program might switch into or from TNS mode or TNS interpreted mode are register-exact points. Contrast with [memory-exact point](#) and [nonexact point](#).

**regular file.** In the Open System Services (OSS) file system, a file that is a randomly accessible sequence of bytes. A regular file contains binary or text data and has no structure imposed by the system. Contrast with [special file](#).

**relative pathname.** In the Open System Services (OSS) file system and Network File System (NFS), a pathname that does not begin with a slash (/) character. A relative pathname is resolved beginning with the current working directory. Contrast with [absolute pathname](#).

**release version update (RVU).** A collection of compatible revisions of HP NonStop Kernel operating system software products, identified by an RVU ID, and shipped and supported as a unit. An RVU consists of the object modules, supporting files, and documentation for the product revisions. An RVU also includes a set of documentation for the overall RVU.

**RELOAD.** An HP Tandem Advanced Command Language (TACL) command to load the HP NonStop™ Kernel operating system image from disk over the ServerNet fabrics into the memory of the processor.

**remote access.** A form of remote support, configured in OSM and TSM Notification Director. Remote access, or dial-in, allows a service provider to dial in to your system

console and access your HP NonStop™ S-series server to diagnose hardware and software problems. See also [remote notification](#).

**remote interprocessor communication (RIPC).** The exchange of messages between processors in different systems or nodes.

**remote mount.** A mount used by a Network File System (NFS) client to attach part of the local NFS file hierarchy to a point within the client's remote file hierarchy. The remote mount is visible only to the NFS client performing the mount. In effect, the local hierarchy from the mount point down is exported to the client performing the remote mount.

**remote node.** See [remote system](#).

**remote notification.** A form of remote support. Remote notification, or dial-out, allows the OSM or TSM Notification Director to notify a service provider, such as the Global Customer Support Center (GCSC), of pending hardware and software problems. See also [remote access](#).

**remote operator.** The person who performs routine system operations from a geographical distance, usually when no local operator is present.

**remote procedure.** A procedure or function packaged to be called within a server process indirectly by a client process.

**remote procedure call.** A remote procedure or the action of calling a remote procedure.

**Remote Procedure Call (RPC).** A protocol that extends a procedure-call form of process-to-process communication to a network environment. RPC is a way for programs running on client computers to invoke the services of a program running on a server computer. RPC allows a program to call a procedure that does not exist on the client computer.

**remote procedure call system.** A set of facilities that includes a programming library, network resource mapping, and binding services to provide a mechanism for a client process to execute a procedure on a remote server. A remote procedure call system is a subset of the Distributed Computing Environment (DCE) and of other products.

**remote processor.** A processor in a node other than the node running the ServerNet cluster monitor process (SNETMON) that is reporting status about the processor.

**remote switch.** An HP NonStop Cluster Switch (for model 6770) or HP NonStop ServerNet Switch (model 6780) in a ServerNet cluster that is not directly connected to the server that you are logged onto. The OSM Service Connection and the TSM Service Application cannot perform any actions on a remote switch. To perform actions or get additional information on a remote switch, use the OSM Service Connection (for either model 6770 or 6780) or TSM Service Application (model 6770 only) to log on to a server that is directly connected to the switch.

**remote system.** An active ServerNet node to which the local system has active external ServerNet paths. Contrast with [local system](#).

**request packet.** A ServerNet packet sent from one ServerNet device to another, requesting either a read action or a write action on the part of the receiving device. In the case of a write request, the packet contains the data to be written. The receiving device is expected to take the appropriate action and return a response packet to the device that sent the request packet. See also [response packet](#), [ServerNet packet](#), and [ServerNet transaction](#).

**reserved symbol.** An identifier that is reserved for use by system or compiler language implementors.

**resistance.** The measure of opposition to current that limits the amount of current that can be produced by an applied voltage. Conductors have very little resistance; insulators have a large amount of resistance. Resistance is measured in ohms.

**resource.** A component of a computer system that works together with other components to process transactions. Terminals, workstations, processors, memory, disk drives, processes, files, and applications are examples of resources.

**response.** The information or confirmation supplied by a subsystem in reaction to a command. A response is typically conveyed as one or more interprocess messages from a subsystem to an application.

**response packet.** A ServerNet packet returned from one ServerNet device to another, responding to an earlier received read request or write request. In the case of responding to a read request, the response packet contains the data that the requesting device wanted to have read. See also [request packet](#), [ServerNet packet](#), and [ServerNet transaction](#).

**RESTORE.** A utility for the HP NonStop™ servers that copies files from a backup tape to disk. See also [BACKUP](#).

**RFC.** Request for Comments. Documents compiled by number by the Internet Engineering Task Force (IETF) that define standards for intercommunication.

**RFI.** See [radio frequency interference \(RFI\)](#).

**RIPC.** See [remote interprocessor communication \(RIPC\)](#).

**RISC.** See [reduced instruction-set computing \(RISC\)](#).

**RISC processor.** An instruction processing unit (IPU) that is based on reduced instruction-set computing (RISC) architecture. TNS/R processors contain RISC processors.

**rld library.** A library that loads position-independent code (PIC) programs and their associated dynamic-link libraries (DLLs). The `rld` library also provides the `dlopen()`, `dlclose()`, `dlresultcode()`, `dlsys()`, and `dlerror()` functions.

**rms.** See [root mean square \(rms\)](#).

**robot.** A media-changer device that transfers an optical disk cartridge from a storage cell to an optical disk drive for use, then returns the cartridge to the storage cell.

**root.** See [root fileset](#) and [root directory](#). See also [super ID](#).

**root directory.** In the Open System Services (OSS) file system and Network File System (NFS), a directory associated with a process that the system uses for pathname resolution when a pathname begins with a slash (/) character.

**root fileset.** For the Open System Services (OSS) file system, the fileset with the device identifier of 0, normally containing the root directory. HP recommends that this fileset be named “root”.

**root mean square (rms).** A measurement method used to determine the direct current (DC) equivalent value for alternating voltage and current waveforms. The rms method refers to the process of sampling a waveform, squaring the samples, averaging the samples (mean value) over the period from one cycle, then calculating the square root of the samples. In general, rms-sensing devices are more accurate than averaging meters. Measurements from averaging meters can be as low as 30 percent of the actual current for loads with high crest factors.

**root user.** See [super ID](#).

**router.** See [ServerNet router](#).

**router 1.** See [ServerNet router 1](#).

**router 2.** See [ServerNet router 2](#).

**RPC.** See [Remote Procedure Call \(RPC\)](#).

**RS-232.** An industry standard for serial data transmission. It describes pin assignments, signal functions, and electrical characteristics. The current standard specifies a 25-pin connector.

**RS-449.** An industry standard for serial data transmission. It specifies pin assignments, signal functions, electrical characteristics, and a 37-pin connector with an optional 9-pin connector for a secondary channel.

**run-time data unit (RTDU).** The region of a TNS object file used to store SQL/MP source and object code. It contains embedded SQL information for clients of SQL/MP. Source RTDUs are created when a program file using embedded SQL/MP is initially compiled and linked. Object RTDUs are added to the program when the file is SQL-compiled.

**run-time linker.** See [linker](#).

**run-time loader.** See [loader](#).

**RVU.** See [release version update \(RVU\)](#).

**S700 server.** See [HP NonStop™ S700 Server](#).

**S7000 server.** See [HP NonStop™ S7000 Server](#).

**S7400 server.** See [HP NonStop™ S7400 Server](#).

**S7600 server.** See [HP NonStop™ S7600 Server](#).

**S70000 server.** See [HP NonStop™ Sxx000 Server](#).

**S72000 server.** See [HP NonStop™ Sxx000 Server](#).

**S74000 server.** See [HP NonStop™ Sxx000 Server](#).

**S76000 server.** See [HP NonStop™ Sxx000 Server](#).

**S86000 server.** See [HP NonStop™ Sxx000 Server](#).

**Sxx000 server.** See [HP NonStop™ Sxx000 Server](#).

**S-series servers.** See [HP NonStop™ S-series servers](#).

**SAC.** See [ServerNet addressable controller \(SAC\)](#).

**sag.** A reduction in voltage, usually lasting from one cycle to a few seconds. Sags are typically caused by fault clearing or by heavy load startups.

**SAN.** System area network. The preferred term is fabrics (see [fabric](#)).

**SANMAN.** See [external system area network manager process \(SANMAN\)](#).

**saveabend file.** A file containing dump information needed by the system debugging tool on a TNS or TNS/R system. In UNIX systems, such files are usually called core files or core dump files. A saveabend file is a special case of a save file. See also [save file](#).

**saved-set group ID.** An Open System Services (OSS) process attribute that stores a group ID so that the group ID can later be used as the effective group ID of the process.

**saved-set user ID.** An Open System Services (OSS) process attribute that stores a user ID so that the user ID can later be used as the effective user ID of the process.

**save file.** A file created through the Inspect or Debug product. A save file contains enough information about a running process at a given time to restart the process at the same

point in its execution. A save file contains an image of the process, data for the process, and the status of the process at the time the save file was created.

A save file can be created through an Inspect SAVE command at any time. A save file called a saveabend file can be created by the DMON debug monitor when a process's SAVEABEND attribute is set and the process terminates abnormally.

**SBB.** See [ServerNet buffer board \(SBB\)](#).

**SBI.** See [ServerNet bus interface \(SBI\)](#).

**SC.** See [Subscriber Channel \(SC\)](#).

**scalar view of the user ID.** A view of the [HP NonStop™ Kernel user ID](#), normally used in the Open System Services (OSS) environment, that is the value  $(group-number * 256) + user-number$ . Also called the [UID](#).

**SCC.** See [serial communications controller \(SCC\)](#).

**SCF.** See [Subsystem Control Facility \(SCF\)](#).

**SCL.** The mnemonic subsystem name for the [ServerNet cluster subsystem](#).

**SCP.** See [Subsystem Control Point \(SCP\)](#).

**SCSI.** See [small computer system interface \(SCSI\)](#).

**SCSI object type.** The Subsystem Control Facility (SCF) object type for an Open SCSI device.

**SCSI passthrough terminator.** A bus-terminating plug connected between a cable and the external connector of a customer-replaceable unit (CRU). The SCSI passthrough terminator contains the necessary termination resistors required by the SCSI bus. See also [terminator](#).

**SCSI plug-in card (S-PIC).** A plug-in card (PIC) for the 6760 ServerNet device adapter (ServerNet/DA) that uses a small computer system interface (SCSI) interface to connect devices to an HP NonStop™ S-series system. See also [plug-in card \(PIC\)](#) and [fiber-optic plug-in card \(F-PIC\)](#).

**SCSI ServerNet addressable controller (S-SAC).** A ServerNet addressable controller (SAC) that is contained within a small computer system interface (SCSI) plug-in card (S-PIC).

**SCSI terminator.** See [terminator](#).

**SE.** System engineer. See [service provider](#).

**searchList.** For each loadfile, a list that is constructed and used by the linker and loader to tell them which libraries to examine, and in which order, to locate symbol definitions

needed by that loadfile. The linker and loader construct the loadfile's searchList in accordance with that loadfile's import control, which is set at link time by the loadfile's programmer. A loadfile's searchList is unaffected by the import control of any other loadfile.

**SEB.** See [ServerNet expansion board \(SEB\)](#).

**SEB port.** A connector on ServerNet expansion boards (SEBs) used for ServerNet links. An SEB has six emitter-coupled logic (ECL)-based ServerNet ports. See also [MSEB port](#).

**Security Manager Process (SMP).** A component of the Safeguard subsystem that manages all changes to the subject and object databases and authenticates user logon attempts.

**segment.** In general, a contiguous sequence of logically-related pages of virtual memory. The pages of the segment are individually swapped in and out of physical memory as needed. Within a loadable object file, one of the portions of the file that will be mapped as one unit into virtual memory as the file is loaded. See also [code segment](#) and [data segment](#).

**selectable segment.** A type of logical segment formerly known as an extended data segment. The data area for a selectable segment always begins with relative segment 4, and this area can be dynamically switched among several selectable segments by calls to the Guardian SEGMENT\_USE\_ procedure; the effect is similar to a rapid overlaying of one large data area. See also [logical segment](#) and [flat segment](#).

**SEM.** See [ServerNet extender module \(SEM\)](#).

**semaphore.** A mechanism used to provide multiple processes with access to a shared data object.

**semi-globalized.** An import control characteristic of a loadfile that allows the loadfile first to obtain symbols from its own definitions and then to obtain others as for a globalized loadfile. See also [searchList](#).

**sensitive command.** A Subsystem Control Facility (SCF) command that can be issued only by a user with super-group access, by the owner of the subsystem, or by a member of the group of the owner of the subsystem. The owner of a subsystem is the user who started that subsystem (or any user whose application ID is the same as the server ID—the result of a PROGID option that requires super-group access). Contrast with [nonsensitive command](#).

**separately derived power source.** A facility wiring system where power is derived from a generator, transformer, or converter windings and there is no direct electrical connection, including a solidly connected grounded circuit conductor (neutral), to supply conductors originating in other facility wiring systems. Types of separately derived power sources include [standby power generator](#), [uninterruptible power supply \(UPS\)](#), [isolation transformer](#), and [computer-room power center \(CRPC\)](#).



**serial communications controller (SCC).** A type of communications controller. Each quad-integrated communications controller (QUICC) has four SCCs to handle the two Ethernet ports and the two wide area network (WAN) ports.

**serial copper.** A standard for physical connectivity in ServerNet I and ServerNet II networks that is available both in HP NonStop™ S-series servers and in Windows NT clusters. Serial copper uses serial encoding and supports 50 and 125 megabyte/second (MB/s) speeds. The maximum link distance at 125 MB/s is 25 meters.

**serial copper PIC.** See [serial copper plug-in card \(PIC\)](#).

**serial copper plug-in card (PIC).** A plug-in card (PIC) for the modular ServerNet expansion board (MSEB) and I/O multifunction (IOMF) 2 customer-replaceable unit (CRU) that supports the serial copper interface. See also [serial copper](#) and [plug-in card \(PIC\)](#).

**serial maintenance bus (SMB).** A bus that connects service processors (SPs) within an enclosure to each other and to the customer-replaceable units (CRUs) in the group.

**serial maintenance bus (SMB) domain.** The set of enclosures, modules, field-replaceable units (FRUs), and customer-replaceable units (CRUs) connected by a common serial maintenance bus (SMB).

**server.** (1) An implementation of a system used as a stand-alone system or as a node in an Expand network. (2) A combination of hardware and software designed to provide services in response to requests received from clients across a network. For example, the HP NonStop™ servers provides transaction processing, database access, and other services. (3) A process or program that provides services to a client or a requester. Servers are designed to receive request messages from clients or requesters; perform the desired operations, such as database inquiries or updates, security verifications, numerical calculations, or data routing to other computer systems; and return reply messages to the clients or requesters. A server process is a running instance of a server program.

**server application.** An application that provides a service to a [client application](#). An application that provides local execution of remote procedure calls is an example of a server application.

**ServerNet.** A communications protocol developed by HP that is used in HP NonStop™ S-series servers. See also [ServerNet I](#) and [ServerNet II](#).

**ServerNet I.** The first-generation ServerNet network. ServerNet I architecture is used in current HP NonStop™ S-series servers and other products, and it features 50 megabytes/second speed, six-port ServerNet routers, 8b/9b encoding, and a 64-byte maximum packet size. See also [ServerNet II](#).

**ServerNet II.** The second-generation ServerNet network. ServerNet II architecture is backward-compatible with ServerNet I architecture, and it features 125 (or 50) megabytes/second speed, 12-port ServerNet routers, 8b/9b and 8b/10b (serializer ready) encoding, and a 512-byte maximum packet size. See also [ServerNet I](#).



**ServerNet adapter.** A component that connects peripheral devices to the rest of the system through a ServerNet bus interface (SBI). A ServerNet adapter is similar in function to an I/O controller logic board (LB) and backplane interconnect card (BIC) in HP NonStop™ K-series servers.

**ServerNet address.** A virtual memory address that, when translated to a physical address, indicates where the memory access needed by a ServerNet transaction begins. In some cases, the translation can point to some entity other than memory, such as a register. The ServerNet address is included in all ServerNet read request and write request packets.

**ServerNet addressable controller (SAC).** An I/O controller that is uniquely addressable by a ServerNet ID in the ServerNet fabrics. A SAC is typically implemented on some portion of a processor multifunction (PMF) customer-replaceable unit (CRU), an I/O multifunction (IOMF) CRU, or a ServerNet adapter.

**ServerNet buffer board (SBB).** The board that provides the ServerNet connection to and from the I/O multifunction (IOMF) customer-replaceable unit (CRU). This board replaces the processor and memory board (PMB) in the IOMF CRU.

**ServerNet bus interface (SBI).** The I/O control and expansion packetizer application-specific integrated circuit (ASIC).

**ServerNet cable.** A cable that provides ServerNet links between system enclosures.

**ServerNet cluster.** A network of servers (nodes) connected together using the ServerNet protocol for interprocessor communication across a cluster and within its nodes. A ServerNet cluster offers linear system expansion beyond the 8-processor or 16-processor limits of a single server, achieving comparable speeds for internal and external ServerNet communication. See also [cluster](#) and [HP NonStop™ ServerNet Cluster \(ServerNet Cluster\)](#).

**ServerNet Cluster.** See [HP NonStop™ ServerNet Cluster \(ServerNet Cluster\)](#).

**ServerNet cluster monitor process (SNETMON).** A process pair with the process name \$ZZSCL that manages the state of the ServerNet cluster subsystem. Each node (system) in a ServerNet cluster must have one SNETMON process pair running.

**ServerNet cluster services.** The functions necessary to allow a node to join, participate in, or leave an HP NonStop™ ServerNet Cluster. These functions include monitoring and control of the physical connections to the cluster, discovery of other nodes in the cluster, and automatic recovery of failed connections.

**ServerNet cluster subsystem.** The [subsystem](#) managed by the [ServerNet cluster monitor process \(SNETMON\)](#). The subsystem name is SCL. The subsystem number is 218. The subsystem identifier is ZSCL.

**ServerNet/DA.** See [ServerNet device adapter \(ServerNet/DA\)](#).

**ServerNet device.** Interface logic that is associated with a specific hardware unit, such as a processor or I/O adapter, and that provides the interface to the ServerNet communications network. The responsibilities of the ServerNet device are to transform message data into ServerNet packets, to transmit those packets, to receive ServerNet packets, and to unpack the data on behalf of the associated hardware unit. See also [ServerNet subdevice](#).

**ServerNet device adapter (ServerNet/DA).** A ServerNet adapter that controls external devices. The 6760 ServerNet/DA contains up to four ServerNet addressable controllers (SACs), each of which can control either disk drives or tape drives.

**ServerNet device ID.** See [ServerNet ID](#).

**ServerNet diagram.** A graphical layout of the logical connections between objects in the system. These objects can include processors, ServerNet routers, ServerNet adapters, disks on the SCSI bus, and so on.

**ServerNet dump.** To copy the memory of a processor to disk by using the ServerNet fabrics.

**ServerNet end device.** See [ServerNet device](#).

**ServerNet expansion board (SEB).** (1) A connector board that plugs in to the backplane to allow one or more ServerNet cables to exit the rear of the enclosure. The SEBs and ServerNet cables allow processors in one group to communicate with processors in another group. Each SEB provides either the ServerNet X fabric or the ServerNet Y fabric for a group. (2) A collective term for both SEBs and modular SEBs (MSEBs) when a distinction between the two types of SEBs is not required.

**ServerNet extender module (SEM).** Equipment that increases the distance that ServerNet signals can be transmitted over fiber-optic cables to 40 kilometers. If multimode fiber-optic (MMF) and single-mode fiber-optic (SMF) ServerNet cables are used in the same system, the SEM converts MMF signals so that they can be transmitted by SMF ServerNet cables and converts SMF signals so that they can be transmitted by MMF ServerNet cables.

**ServerNet/FX adapter.** A ServerNet adapter that logically extends the ServerNet X and Y fabrics to other clusters in a Fiber Optic Extension (FOX) ring by using fiber-optic lines. Two 6740 ServerNet/FX adapters are used, one for the X ring and one for the Y ring.

**ServerNet/FX 2 adapter.** A ServerNet adapter that logically extends the ServerNet X and Y fabrics to other clusters in a Fiber Optic Extension (FOX) ring by using fiber-optic lines. Two 6742 ServerNet/FX 2 adapters are used, one for the X ring and one for the Y ring.

**ServerNet ID.** A unique identifier for an addressable unit on a ServerNet communications network. A unit can have multiple ServerNet node IDs. This ID is used for routing: each packet has a source ServerNet node ID and a destination ServerNet node ID. Note that a pair of processors operating in duplex mode share one ServerNet node ID.

**ServerNet LAN Systems Access (SLSA) subsystem.** A subsystem of the HP NonStop™ Kernel operating system for configuration and management of ServerNet local area network (LAN) objects in G-series release version updates (RVUs).

**ServerNet link.** Two unidirectional point-to-point communication paths, one in each direction, connecting a router to a ServerNet node or another router. Each ServerNet link contains a transmit channel and a receive channel.

**ServerNet memory interface (SMI).** An application-specific integrated circuit (ASIC) that provides the interface between the microprocessor and the two ServerNet fabrics and main memory.

**ServerNet node.** A system in a [ServerNet cluster](#). See also [node](#).

**ServerNet node number.** A number that identifies a member system in a ServerNet cluster. The ServerNet node number is a simplified expression of the 6-bit node-routing ID that determines the node to which a ServerNet packet is routed. The ServerNet node number is assigned based on the port to which the node is connected on the cluster switch. The ServerNet node number, which can be viewed using the Subsystem Control Facility (SCF), the OSM Service Connection, or the TSM Service Application, is unique for each node in a ServerNet cluster.

**ServerNet node routing ID.** A bit field used to route ServerNet packets across the external ServerNet X and Y fabrics. The ServerNet node routing ID occupies the upper six bits of the 20-bit ServerNet ID, and it is unique for each member, or node, in a ServerNet cluster. This term is the fully qualified form of “node routing ID.”

**SERVERNET object type.** In the Kernel subsystem, the object type for either the \$ZSNET ServerNet subsystem manager process or the ServerNet X fabric or Y fabric.

**ServerNet packet.** The unit of transmission in a ServerNet communications network. A ServerNet packet consists of a header, a variable-size data field, and a 32-bit cyclic redundancy check (CRC) checksum covering the entire packet. The header contains fields for control, virtual memory address, and destination and source fields to identify the processor or I/O controller transmitting and receiving the packet. See also [request packet](#) and [response packet](#).

**ServerNet port.** A connector used for ServerNet links. Six ServerNet ports are located on a ServerNet expansion board (SEB). Ten ServerNet ports are located on a modular ServerNet expansion board (MSEB).

**ServerNet router.** An application-specific integrated circuit (ASIC) responsible for routing ServerNet packets along ServerNet links in the ServerNet fabrics, using routing information that is present within the packets. A ServerNet router acts as a fully duplex crossbar switch, able to switch any of its input ports to any of its output ports. A ServerNet router in an HP NonStop™ S-series server has either six router ports (see [ServerNet router 1](#)) or twelve router ports (see [ServerNet router 2](#)).

**ServerNet router 1.** A model of ServerNet router that, in an HP NonStop™ S-series server, has a total of six input and six output ports. See also [ServerNet router 2](#).

**ServerNet router 2.** A model of ServerNet router that, in an HP NonStop™ S-series server, has a total of twelve input and twelve output ports. See also [ServerNet router 1](#).

**ServerNet subdevice.** An I/O device that sends and receives its information through a controlling device that acts as the ServerNet device for routing purposes.

**ServerNet subdevice ID.** The low-order (least significant) bits of a ServerNet ID, used by a ServerNet device to distribute incoming and outgoing information between itself and its associated subdevices.

**ServerNet switch.** A point-to-point networking device that connects ServerNet nodes to a single fabric (X or Y) of the ServerNet communications network. The ServerNet switch routes ServerNet packets among these nodes.

**ServerNet II Switch.** A 12-port network switch that provides the physical junction point to enable an HP NonStop™ S-series server to connect to a ServerNet cluster. The ServerNet II Switch is a component of the [HP NonStop™ Cluster Switch \(model 6770\)](#).

**ServerNet transaction.** The bidirectional, successful transmission of a pair of ServerNet packets between two ServerNet devices. The device that originates the transaction sends a request packet, and the device that receives the request returns a response packet. See also [request packet](#) and [response packet](#).

**ServerNet wide area network (SWAN) concentrator.** (1) An HP data communications peripheral that provides connectivity to an HP NonStop™ S-series server. The SWAN concentrator supports both synchronous and asynchronous data over RS-232, RS-449, X.21, and V.35 electrical and physical interfaces. (2) A collective term for both SWAN concentrators and SWAN 2 concentrators when a distinction between the two is not required.

**ServerNet wide area network (SWAN) 2 concentrator.** An HP data communications peripheral that provides connectivity to an HP NonStop™ S-series server. The SWAN 2 concentrator supports both synchronous and asynchronous data over RS-232, RS-449, X.21, and V.35 electrical and physical interfaces. The SWAN 2 concentrator is the next-generation SWAN concentrator and has 12 WAN ports.

**service connection.** A connection between the Compaq TSM client software running on a system console and the TSM server software running on an HP NonStop™ S-series server. A service connection can be used only to communicate with the server when the HP NonStop™ Kernel operating system is running. A service connection provides a comprehensive service and maintenance picture of the server and is used to perform most service management tasks. See also [low-level link](#).

**service equipment.** The necessary equipment, usually consisting of circuit breakers and their accessories, that is located near the entrance point of supply conductors. This equipment constitutes the main control and cutoff means of the supply.

**service processor (SP).** A physical component of the processor multifunction (PMF) customer-replaceable unit (CRU) or I/O multifunction (IOMF) CRU that controls environmental and maintenance functions (including system load functions) in the enclosure. SPs operate in pairs to provide fault tolerance. The two SPs in group 01 are designated the master service processors (MSPs). Other pairs of SPs within a system are called expansion service processors (ESPs). See also [expansion service processor \(ESP\)](#) and [master service processor \(MSP\)](#).

**service provider.** (1) A person trained and qualified to service field-replaceable units (FRUs). (2) An organization, such as the Global Customer Support Center (GCSC), that helps you resolve problems with your HP NonStop™ S-series server. OSM and TSM allow you to use the help of a service provider by configuring client software to support remote notification and remote access.

**service side.** The side of a system enclosure that contains, behind an optional door, processor multifunction (PMF) customer-replaceable units (CRUs) or I/O multifunction (IOMF) CRUs, ServerNet expansion boards (SEBs), modular SEBs (MSEBs), and ServerNet adapters; it is opposite the [appearance side](#). Cables are accessed from the service side. System enclosures are typically arranged so that the service side is the least visible side.

**session.** In the Open System Services (OSS) environment, a set of process groups associated for job control purposes. A session can have a controlling terminal.

**session leader.** In the Open System Services (OSS) environment, the process that created a session.

**session lifetime.** In the Open System Services (OSS) environment, the period that begins when a session is created and ends when the lifetime of the last remaining process group of the session ends.

**setup configuration.** A simple stand-alone network used to configure the OSM or TSM environment. The setup configuration consists of the server, the primary system console, an Ethernet hub, and two local area network (LAN) cables. One cable connects the primary system console to the hub, and another cable connects the hub to a processor multifunction (PMF) customer-replaceable unit (CRU) in group 01 of the server. When OSM or TSM configuration is complete, the setup configuration can serve as the working network configuration. Variations of the setup configuration can be constructed using additional cables and hubs to increase fault tolerance.

**set-user-ID program.** In the Open System Services (OSS) environment, a program file that has the `S_ISUID` bit set in its file mode.

**shared memory.** An interprocess communication mechanism that allows two or more processes to share a given region of memory.

**Shared Millicode Library.** An intrinsic library containing privileged or TNS-derived millicode routines used by many native-compiled programs and by emulated TNS programs. This library includes efficient string-move operations, TNS floating point emulation, and

various privileged-only operations. These routines are mode-independent; they comply with native calling conventions but can be directly invoked from any mode without changing execution modes.

**shared run-time library (SRL).** A collection of procedures whose code and data can be loaded and executed only at a specific assigned virtual memory address (the same address in all processes). SRLs use direct addressing and do not have run-time resolution of links to and from the main program and other independent libraries. Contrast with [dynamic-link library \(DLL\)](#). See also [TNS shared run-time library \(TNS SRL\)](#) and [TNS/R native shared run-time library \(TNS/R native SRL\)](#).

**shell.** In the Open System Services (OSS) environment, a program that interprets sequences of text input as commands. A shell can operate on an input stream, or it can interactively prompt and read commands from a terminal.

**shielded twisted pair (STP).** A transmission medium consisting of two twisted conductors with a foil or braid shield. Contrast with [unshielded twisted pair \(UTP\)](#).

**shutdown file.** A file invoked by the local operator or by another shutdown file that contains commands to shut down system devices, communications lines, and system and application software. Contrast with [startup file](#).

**SID.** See [system image disk \(SID\)](#) or [source ServerNet ID \(SID\)](#).

**signal.** The method by which an environment notifies a process of an event. Signals are used to notify a process when an error that is not related to input or output has occurred. See also [Open System Services \(OSS\) signal](#).

**signal delivery.** The time when Open System Services (OSS) takes the action appropriate for a specific process and a specific signal.

**signal generation.** The time when an event occurs that causes a signal for a process.

**signal handler.** A function or procedure that is executed when a specific signal is delivered to a specific process.

**Signaling System Number 7 (SS7).** The protocol used in public networks to establish connections between switches.

**signal mask.** The set of signals that are currently blocked from delivery to a specific process.

**signal reference grid.** A series of conductors, constructed of pure or composite metals (for example, copper) with good surface conductivity. A superior signal reference grid is installed on the subfloor of a computer room and connected to the raised-floor structure to establish constant and equal potential for all equipment in the computer room that is connected to it.



**Simple Network Management Protocol (SNMP).** An asynchronous request-response protocol used for network management. SNMP originated as a means for managing Transmission Control Protocol/Internet Protocol (TCP/IP) and Ethernet networks. OSM and TSM packages can include an SNMP-compliant interface for communication between the system console and HP NonStop™ S-series server.

**single-high ServerNet adapter.** A ServerNet adapter that occupies only the upper half of a ServerNet adapter slot in an HP NonStop™ S-series server. Contrast with [double-high ServerNet adapter](#).

**single-high stack.** A stack that includes a base, a frame, and one system enclosure. Contrast with [double-high stack](#).

**single-mode fiber-optic (SMF) plug-in card (PIC).** A plug-in card (PIC) for the modular ServerNet expansion board (MSEB) and I/O multifunction (IOMF) 2 customer-replaceable unit (CRU) that supports the single-mode fiber-optic (SMF) interface.

**single-mode fiber-optic (SMF) ServerNet cable.** A fiber-optic cable that allows only one mode to propagate. SMF ServerNet cable has a small-diameter core for optimized long-distance transmission. See also [multimode fiber-optic \(MMF\) ServerNet cable](#).

**single-wide plug-in card (PIC).** A small-form-factor [plug-in card \(PIC\)](#) that occupies one PIC slot within a customer-replaceable unit (CRU). See also [double-wide plug-in card \(PIC\)](#).

**sinusoidal.** A waveform that can be mathematically expressed by the sine function.

**SIT.** See [system image tape \(SIT\)](#).

**site update tape (SUT).** One or more tapes that contain each target system's site-specific subvolume and various products. Each product contains a softdoc and a complete set of files. A SUT is delivered with every new HP NonStop™ S-series system and can be ordered whenever a new release version update (RVU) of the system software is available. A full SUT contains the current RVU of the HP NonStop Kernel operating system and all product software that has been ordered with it. A partial SUT contains a subset of products for the current software RVU.

**SIV.** See [system interrupt vector \(SIV\)](#).

**skin effect.** The tendency of higher frequency signals to flow on the outside surface, or skin, of a conductor instead of through the entire cross-section of the conductor. The result is less total conductor area available for carrying current and an increase in the resistance of the conductor at that high signal frequency.

**slot.** A physical, labeled space for a customer-replaceable unit (CRU) or field-replaceable unit (FRU) that is part of a module. A module contains one or more slots.

**slot location.** A three-number identifier for a particular slot on a system that consists of the group number, module number, and slot number; for example, 02,01,08 (group 02, module 01, slot 08).

**SLSA subsystem.** See [ServerNet LAN Systems Access \(SLSA\) subsystem](#).

**small computer system interface (SCSI).** An ANSI-standard protocol used by a controller to access a device.

**SMB.** See [serial maintenance bus \(SMB\)](#).

**SMB domain.** See [serial maintenance bus \(SMB\) domain](#).

**SMF.** See [Storage Management Foundation \(SMF\)](#).

**SMF PIC.** See [single-mode fiber-optic \(SMF\) plug-in card \(PIC\)](#).

**SMF ServerNet cable.** See [single-mode fiber-optic \(SMF\) ServerNet cable](#).

**SMI.** See [ServerNet memory interface \(SMI\)](#).

**SMN.** The mnemonic name for the [external system area network manager process \(SANMAN\)](#).

**SMP.** See [Security Manager Process \(SMP\)](#).

**SMT.** See [SWAN manager task \(SMT\)](#).

**snapshot.** (1) A file that can be created by OSM and TSM client software to record information about the status of an HP NonStop™ S-series server, including the attributes values of all system resources, at the time it was created. The file can be forwarded to your service provider to help with troubleshooting problems. (2) For Distributed Systems Management/Software Configuration Manager (DSM/SCM), a list of the target system tape and disk locations, file fingerprints for files managed by DSM/SCM, and DSM/SCM target information. The snapshot is compiled on the target system from the target database and sent to the host system to store in the host database. An instruction to create a snapshot is part of every activation package sent from the host and can also be requested independently through the Target Interface.

**SNDA.** See [ServerNet device adapter \(ServerNet/DA\)](#).

**SNETMON.** See [ServerNet cluster monitor process \(SNETMON\)](#).

**SNMP.** See [Simple Network Management Protocol \(SNMP\)](#).

**SNMP task.** A task that runs in each ServerNet wide area network (SWAN) concentrator communications line interface processor (CLIP) as part of the WAN architecture. This task accepts and replies to Simple Network Management Protocol (SNMP) request messages.



**socket.** An end-point for stream-oriented communication. A socket has a file descriptor.

**soft reset.** An action performed on an HP NonStop™ Cluster Switch that restarts the firmware on the cluster switch but does not interfere with ServerNet passthrough data traffic.

**software configuration incident report.** A type of incident report that reports changes in the software configuration of the server. A software configuration incident report includes the server's software configuration file.

**software product revision (SPR).** The method of releasing incremental software updates on HP NonStop™ S-series systems. An SPR can include one or more corrections to code or it can contain code that adds new function to a software product.

**source ServerNet ID (SID).** A field in the ServerNet packet header indicating the source of the packet.

**SP.** See [service processor \(SP\)](#).

**special character.** A character entered from a terminal that has an effect other than being part of the input stream from that terminal.

**special file.** A file in the Open System Services (OSS) file system that is not a regular file. Special files include directories, FIFOs, and character special files such as terminal device files. Contrast with [regular file](#).

**SP event message.** See [service processor \(SP\)](#).

**SPI.** See [Subsystem Programmatic Interface \(SPI\)](#).

**S-PIC.** See [SCSI plug-in card \(S-PIC\)](#).

**split-star topology.** A network [topology](#) that uses up to two HP NonStop™ Cluster Switches for each external fabric. External routing is implemented between the two starred halves of a ServerNet cluster. (A starred half consists of up to eight nodes attached to one set of cluster switches.) The starred segments are joined by [four-lane links](#). Introduced with the G06.12 release version update (RVU) of the HP NonStop ServerNet Cluster product, the split-star topology supports up to 16 nodes. See also [tri-star topology](#).

**spooler.** The collection of files and processes that manages the printers and print jobs on the system.

**SPR.** See [software product revision \(SPR\)](#).

**SP Tool Application.** A PC-based software application that you can use to request information from the master service processors (MSPs) in an HP NonStop™ S-series server. This application is intended for use only by trained service providers.

**SRL.** See [shared run-time library \(SRL\)](#).

**SRM.** See [system resource model \(SRM\)](#).

**SS7.** See [Signaling System Number 7 \(SS7\)](#).

**SS7TE PIC.** See [SS7TE plug-in card \(PIC\)](#).

**SS7TE2 PIC.** See [SS7TE2 plug-in card \(PIC\)](#).

**SS7TE plug-in card (PIC).** A plug-in card (PIC) used in the 6763 Common Communication ServerNet adapter (CCSA) that supports the EIA-232, EIA-449, V.35, and X.21 interfaces.

**SS7TE2 plug-in card (PIC).** A plug-in card (PIC) used in the 6763 Common Communication ServerNet adapter (CCSA) that supports the E1, J1, and T1 interfaces.

**S-SAC.** See [SCSI ServerNet addressable controller \(S-SAC\)](#).

**SSI log.** See [System Service Information \(SSI\) log](#).

**stackable enclosure.** An enclosure that can rest on top of another enclosure. A stackable enclosure is not installed on a frame base. Contrast with [base enclosure](#).

**standby power generator.** A turbine-driven or engine-driven generator that provides a backup source of power to designated loads. Often used to supplement an [uninterruptible power supply \(UPS\)](#) in the event of extended utility outages.

**star group.** One set of X and Y HP NonStop™ Cluster Switches and the ServerNet nodes (up to eight) that are connected to them. A star group can be thought of as a segment of a split-star or tri-star topology. A split-star topology can contain up to two star groups; a tri-star topology can contain up to three star groups.

**start mode.** An attribute of Subsystem Control Facility (SCF) PROCESS objects that controls when and if an application process starts in G-series release version updates (RVUs).

**star topology.** A network [topology](#) in which all nodes are connected to a central hub (HP NonStop™ Cluster Switch). Each node has its own connection to the network, so a break in the connection does not affect other nodes in the network. In a ServerNet cluster, a star topology requires one cluster switch for each external fabric and can support up to eight nodes. See also [split-star topology](#) and [tri-star topology](#).

**startup file.** A file invoked by the local operator or by another startup file that contains commands to start up system devices, communications lines, and system and application software. Contrast with [shutdown file](#).

**state.** In Subsystem Control Facility (SCF), one of the generally defined possible conditions of an object with respect to the management of that object. Examples of states are DEFINED, STARTED, and STOPPED.

**static information.** Information that represents the set of customer-replaceable units (CRUs) on an HP NonStop™ S-series system. Contrast with [dynamic information](#).

**static server.** In the Guardian environment, a process that runs continuously and provides a specific service to other processes. A static server differs from a traditional UNIX demon in that a demon actively looks for tasks to perform, while a static server performs only tasks brought to its attention by a client (requestor) process. See also [demon](#).

**STFs.** See [super time factors \(STFs\)](#).

**storage cell.** A compartment in an [optical storage library \(OSL\)](#) that stores an [optical disk cartridge](#).

**Storage Management Foundation (SMF).** A subsystem used by the storage subsystem that facilitates automation of storage management tasks by providing location-independent naming, storage pools, and virtual disks on HP NonStop™ S-series systems.

**storage pool.** A set of physical disk volumes administered as a set of logical disk volumes. A logical disk volume can span multiple physical disk volumes. When a logical disk volume becomes full, more physical disk volumes can be added.

**storage-pool file.** A file containing a list of disk volumes to be used by an Open System Services (OSS) fileset. As these volumes are filled, more volumes can be added to the storage-pool file.

**storage subsystem.** A subsystem of the HP NonStop™ Kernel operating system that handles configuration and management of disk and tape devices in G-series release version updates (RVUs).

**storage subsystem manager process.** The generic process that starts and manages disk and tape drives. The \$ZZSTO storage subsystem manager process is started and managed by the \$ZZKRN Kernel subsystem manager process through the \$ZPM persistence manager process.

**store and forward routing.** A form of message routing whereby a router must receive an entire packet or message before it can start to forward the packet or message to the next router. Contrast with [wormhole routing](#).

**STP.** See [shielded twisted pair \(STP\)](#).

**strictly conforming POSIX.1 application.** An application that requires only the facilities described in ISO/IEC IS 9945-1:1990 and the applicable computer language standards. Such an application must accept any behavior or value described in

ISO/IEC IS 9945-1:1990 as unspecified or implementation-defined and, for symbolic constants, accept any value permitted by ISO/IEC IS 9945-1:1990.

**structured view of the user ID.** A view of the [HP NonStop™ Kernel user ID](#), normally used in the Guardian environment, that consists of either the *group-number*, *user-number* pair of values or the *group-name*.*user-name* pair of values.

**subnet.** See [subnetwork](#).

**subnetwork.** A physical network within an Internet protocol (IP) network. Each IP network can be divided into a number of subnetworks. Within a given network, each subnetwork is treated as a separate network. Outside the network, the subnetworks appear as part of a single network. The terms subnetwork and subnet are used interchangeably.

**subnetwork address.** An extension of the Internet protocol (IP) addressing scheme that allows a site to use a single IP address for multiple physical networks. A subnetwork address is created by dividing the local part of an IP address into a subnetwork number (identifying a particular subnetwork) and a host number (uniquely identifying the host system within the subnetwork). The terms subnetwork address and subnet address are used interchangeably.

**SUB option.** In some Subsystem Control Facility (SCF) subsystems, the designation that the object name given in a command stands not just for itself, but for the names of all objects at the next-lower level in the hierarchy. The given object name can stand both for itself and for the subordinate objects, or it can stand only for the subordinate objects, depending on the value of the SUB option.

**subordinate objects.** In Subsystem Control Facility (SCF), objects that are logically subordinate to other objects. Some subsystems are structured hierarchically, with objects of one type logically subordinate to (that is, controlled by) an object of another type. For example, a number of subdevices can be configured on a single line. Some SCF commands include a SUB option that refers to subordinate objects.

**Subscriber Channel (SC).** A type of head on a fiber-optic cable in which the pins connect through a push-pull mating interface.

**substate.** Further information about the state of a device. The state and substate together provide information about the current condition of a device or path to a device.

**SUBSYS object type.** The Subsystem Control Facility (SCF) object type for most subsystems that use SCF as the user interface.

**subsystem.** (1) A secondary or subordinate system, usually capable of operating independently of or asynchronously with a controlling system. (2) A program or set of processes that manages a cohesive set of Subsystem Control Facility (SCF) objects. Each subsystem has a manager through which applications can request services by issuing commands defined by that subsystem. See also [subsystem manager](#).

**Subsystem Control Facility (SCF).** An interactive interface for configuring, controlling, and collecting information from a subsystem and its objects. SCF enables you to configure and reconfigure devices, processes, and some system variables while your HP NonStop™ S-series server is online.

**Subsystem Control Point (SCP).** The message router for Subsystem Control Facility (SCF). There can be several instances of this process. Using the Subsystem Programmatic Interface (SPI), applications send each command for a subsystem to an instance of the SCP process, which in turn sends the command to the manager process of the target subsystem. SCP also processes a few commands itself. It provides security features, version compatibility, support for tracing, and support for applications implemented as process pairs.

**subsystem manager.** A process that performs configuration and management functions for a subsystem.

**Subsystem Programmatic Interface (SPI).** A set of procedures and associated definition files and a standard message protocol used to define common message-based interfaces for communication between management applications and subsystems. It includes procedures to build and decode specially formatted messages; definition files in Transaction Application Language (TAL), COBOL85, and HP Tandem Advanced Command Language (TACL) for inclusion in programs, macros, and routines using the interface procedures; and definition files in Data Definition Language (DDL) for programmers writing their own subsystems.

**subvolume.** A group of related files stored on a disk; all the files have the same volume and subvolume name, but each file has a unique file identifier.

**summary report.** A brief informational listing of status or configuration information provided by the Subsystem Control Facility (SCF) STATUS or INFO command. Contrast with [detailed report](#).

**superblock.** The part of the Open System Services (OSS) environment that contains all the information about the current state of the OSS file system. The superblock contains such items as the free list and the size of inodes.

**super group.** The group of user IDs that have 255 as the group number. This group has special privileges; many HP utilities have commands or functions that can be executed only by a member of the super group.

**super-group user.** A user who can read, write, execute, and purge most files on the system. Super-group users have user IDs that have 255 as the group number.

**super ID.** On HP NonStop™ systems, a privileged user who can read, write, execute, and purge all files on the system. The super ID is usually a member of a system-supervisor group.

The super ID has the set of special permissions called appropriate privileges. In the Guardian environment, the structured view of the super ID, which is (255, 255), is most

commonly used; in the Open System Services (OSS) environment, the scalar view of the super ID, which is 65535, is most commonly used.

**super time factors (STFs).** An enhancement to the Expand product that allows the extension of automatically calculated time factors to line speeds greater than 224 kilobits/second. These time factors are logarithmic-based and allow specification of a much broader range of line performance.

**superuser.** See [super ID](#).

**supplementary group ID.** An Open System Services (OSS) process attribute that is used to determine the file access permissions for the process.

**support planner.** The person who creates the operational environment for the system and is responsible for the support of the system. This person creates the startup and shutdown files, performs replacement operations, and prepares the system for upgrades and additions.

**surge.** An increase in the amplitude of source voltage of short duration.

**SUT.** See [site update tape \(SUT\)](#).

**SVID.** The *System V Interface Definition* for UNIX, published by AT&T.

**SVR4.** System V Release 4, a specific implementation of UNIX. See also [System V](#).

**SWAN concentrator.** See [ServerNet wide area network \(SWAN\) concentrator](#).

**SWAN 2 concentrator.** See [ServerNet wide area network \(SWAN\) 2 concentrator](#).

**SWAN manager task (SMT).** A manager task that is provided as part of the wide area network (WAN) subsystem. The SMT runs in each communications line interface processor (CLIP) and provides a variety of management functions such as coordinating data link control (DLC) and diagnostic task downloads.

**SWID.** The software identification tool invoked by the SYSGENR program that audits file identification information about your software.

**switch mode power supply.** A computer power supply that uses a pulse-width modulation switching inverter and nonlinear current draw characteristics. Switch-mode power supplies are widely used because of their small size and efficiency.

**symbol.** (1) The symbolic name of a value, typically a function entry point or a data location. In the context of loadable libraries, symbols are defined in loadfiles and referenced in the same or other loadfiles. (2) Within the ServerNet architecture, the nine or more bits that encode 8-bit data and protocol commands.

**symbolic link.** In the Open System Services (OSS) file system and Network File System (NFS), a type of special file that acts as a name pointer to another file. A symbolic link



contains a pathname and can be used to point to a file in another fileset. Symbolic links are not included in ISO/IEC IS 9945-1:1990. Contrast with [hard link](#).

**symbolic reference.** An occurrence in code or data of the value of a symbol. The symbolic reference is bound (resolved and made usable) by assigning to it the value of a definition of that symbol. The symbol value is normally the address of a function or variable named by the symbol. In position-independent code (PIC) loadfiles, symbolic references occur only in data.

**symbols region.** See [Inspect region](#).

**SYSGENR.** The system generation program that generates a customized version of the HP NonStop™ Kernel operating system.

**SYS<sub>nn</sub> subvolume.** A subvolume on the \$SYSTEM volume where the new version of the HP NonStop™ Kernel operating system image is located. Also located on the SYS<sub>nn</sub> subvolume is system-dependent and release version update (RVU)-dependent software. *nn* is an octal number in the range %00 through %77.

**SYSPOOL.** The system data space that remains in memory after all system data structures are built by the HP NonStop™ Kernel operating system at the time of a processor load.

**system.** All the processors, controllers, firmware, peripheral devices, software, and related components that are directly connected together to form an entity that is managed by one HP NonStop™ Kernel operating system image and operated as one computer. See also [node](#).

**system area network management process.** See [external system area network manager process \(SANMAN\)](#).

**System Code (SC).** See [TNS code space](#).

**system code.** A logically distinct part of the HP NonStop™ Kernel operating system that consists of operating-system procedures shared by all processors.

**system configuration database.** The database file on the \$SYSTEM.ZSYSCONF subvolume that contains configuration information for all system objects that can be configured by the Subsystem Control Facility (SCF). Configuration information for all system objects that can be configured during system generation is contained on the \$SYSTEM.SYS<sub>nn</sub> subvolume. See also [configuration file](#) and [SYS<sub>nn</sub> subvolume](#).

**system console.** An HP-approved personal computer used to run maintenance and diagnostic software for HP NonStop S-series servers. New system consoles are preconfigured with the required HP and third-party software. When upgrading to the latest RVU, software upgrades can be installed from the HP NonStop System Console Installer CD. System consoles communicate with NonStop S-series servers over a dedicated local area network (LAN) or a nondedicated (public) LAN. System consoles configured as the primary and backup dial-out points are referred to as the primary and backup system consoles, respectively.

**system enclosure.** An enclosure for system components. Processor enclosures and I/O enclosures are both system enclosures. Contrast with [peripheral enclosure](#).

**system engineer (SE).** See [service provider](#).

**system entry point table (SEP table).** A table used on TNS systems that stores the XEP entry value for each TNS operating system procedure entry point.

**system expansion.** The process of making a target system larger by adding enclosures to it. The enclosures being added can be either new enclosures or enclosures from a donor system. Contrast with [system reduction](#).

**system generation.** The process of creating an operating system to support a particular system configuration and software release version update (RVU).

**system image disk (SID).** A disk copy of the HP NonStop™ Kernel operating system produced during system configuration. The DISKGEN utility copies operating system files to the SID.

**system image tape (SIT).** A tape that can be used to perform a system load on a system if the system subvolume has become corrupted on both \$SYSTEM disks. The tape contains a minimum set of software necessary to bring up and run the system. Use the SIT only for disaster recovery; it is not needed for normal system load. Contact the Global Customer Support Center (GCSC) before loading the system from a SIT; many additional steps are required to restore your system to working order. See also [tape load](#).

**system interrupt vector (SIV).** A HP NonStop™ Kernel operating system data structure that contains the addresses of interrupt handlers, parameters passed to interrupt handlers by special interrupt microcode, and other interrupt processing information.

**System Library (SL).** See [TNS code space](#).

**system library.** A logically distinct part of the HP NonStop™ Kernel operating system that consists of user-callable library procedures and kernel procedures.

**system load.** (1) To start the system; to load the HP NonStop™ Kernel operating system image into the memory of a processor. See [RELOAD](#). (2) The process of loading the operating system. A system load changes a system from an inactive to an active (or operational) state by loading software that establishes communication between the operating system and configured system peripherals.

**system-managed process.** Another name for a [generic process](#).

**system manager.** See [manager](#).

**system number.** See [Expand node number](#).

**system operator.** See [local operator](#) and [remote operator](#).



**system planner.** The person who plans for the hardware and software installation of a new system or for changes to a system already installed. This person arranges for site preparation, schedules the installation, and completes the Installation Document Packet.

**system process.** (1) A privileged process that comes into existence at system-load time and exists continuously for a given configuration for as long as the processor remains operable. (2) A HP NonStop™ Kernel operating system process, such as the memory manager, the monitor, and the input/output (I/O) control processes. The files containing system processes are invoked by ALLPROCESSORS paragraph entries. (3) A part of a single copy of the HP NonStop Kernel operating system with Open System Services (OSS) interfaces. A system process does not have an OSS process ID.

**system reduction.** The process of making a donor system smaller by removing enclosures from it. The enclosures removed from a donor system might be added to a target system. Contrast with [system expansion](#).

**system resizing.** See [system expansion](#) or [system reduction](#).

**system resource model (SRM).** A collection of C++ objects that model the diagnostic and serviceability state behavior of the system resources discovered and managed by the Compaq TSM package. The SRM has the following attributes:

Generic process name	\$ZZKRN.#TSM-SRM
Process name	\$ZTSM
Program file name	\$SYSTEM.SYS <sub>nn</sub> .SRM

**system serial number.** A unique identifier, typically five or six alphanumeric characters, assigned to an HP NonStop™ S-series server when it is built.

**System Service Information (SSI) log.** An Event Management Service (EMS) log that includes information about customer-replaceable unit (CRU) removal, CRU insertion, firmware loading, security authentication, incident report dial-out authorization, incident report dial-out failure, and incident report confirmation. The SSI log can be viewed using the OSM or the TSM Event Viewer.

**system terminal.** See [system console](#).

**System V.** A version of UNIX developed and marketed originally by AT&T.

**TACL.** See [HP Tandem Advanced Command Language \(TACL\)](#).

**tape bootstrap.** The program on a system image tape (SIT) that reads the rest of the SIT during tape load and writes it to the system disk.

**tape drive.** A device that moves magnetic tape past magnetic read/write heads, which read data from or write data to the tape.

**tape dump.** To copy the memory of a processor to tape.

**tape library.** A storage device consisting of magnetic tape drives, multiple storage locations for magnetic tape cartridges, an automatic mechanism for loading the cartridges into and unloading them from the drives, and a means for an operator to load cartridges into or remove cartridges from the tape library.

**tape load.** A system load. A tape load is the process of reading a system image tape (SIT) and writing it to the system disk. Performing a tape load from a SIT to restore the system image files to the \$SYSTEM disk is generally not recommended. Perform a tape load only with the advice of the Global Customer Support Center (GCSC) or your service provider. Loading from a tape reinitializes the disk directory.

**TAPE object type.** The Subsystem Control Facility (SCF) object type for all tape drives attached to your system.

**target system.** The computer system you make larger by adding enclosures, using a process known as system expansion. See also [donor system](#).

**TB.** See [terabyte \(TB\)](#).

**TCP.** See [Transmission Control Protocol \(TCP\)](#).

**TCP/IP.** See [Transmission Control Protocol/Internet Protocol \(TCP/IP\)](#).

**TEMPLI.** The Event Management Service (EMS) template installation program that merges template object files from specified subsystems and produces resident and nonresident template files.

**terabyte (TB).** A unit of measurement equal to 1,099,511,627,776 bytes (1024 gigabytes). See also [gigabyte \(GB\)](#), [kilobyte \(KB\)](#), and [megabyte \(MB\)](#).

**terminal.** A type of Open System Services (OSS) character special file that conforms to the interface description in Clause 7 of ISO/IEC IS 9945-1:1990.

**terminator.** A resistor connected to a signal wire in a bus or network for the purpose of impedance matching to prevent reflections. SCSI chains, Ethernet cables, and some LocalTalk wiring configurations require terminators.

**Tetra 8 topology.** A [tetrahedral topology](#) of HP NonStop™ S-series servers that allows a maximum of four processor enclosures (eight processors) and eight I/O enclosures. Contrast with [Tetra 16 topology](#).

**Tetra 16 topology.** A [tetrahedral topology](#) of HP NonStop™ S-series servers that allows a maximum of eight processor enclosures (16 processors). The maximum number of I/O enclosures allowed by the Tetra 16 topology varies depending on the software release version update (RVU) and the server model. Contrast with [Tetra 8 topology](#).

**tetrahedral topology.** A topology of HP NonStop™ S-series servers in which the ServerNet connections between the processor enclosures form a tetrahedron. See also [tetrahedron](#) and [topology](#).

**tetrahedron.** A solid bounded by four triangular faces. In ServerNet context, a tetrahedron is four processors interconnected by ServerNet links so as to form a conceptual tetrahedron. Each processor therefore has a direct connection to the other three processors. See [tetrahedral topology](#).

**text string.** A variable-length sequence of ASCII characters, defined in the CONFTEXT file, that an [identifier](#) represents. When Distributed Systems Management/Software Configuration Manager (DSM/SCM) encounters an identifier, it substitutes the associated text string for the identifier.

**TF.** See [time factor \(TF\)](#).

**TFDS.** See [HP Tandem Failure Data System \(TFDS\)](#).

**TFTP.** See [Trivial File Transfer Protocol \(TFTP\)](#).

**THD.** See [total harmonic distortion \(THD\)](#).

**three point fall of potential measurement method.** The measurement of a grounding electrode (such as a ground rod) where ground resistance is measured with respect to two other points. The ratio of the measurements determines the resistance of the grounding electrode.

**TIM.** See [Total Information Manager \(TIM\)](#).

**time factor (TF).** A number assigned to a line, path, or route to indicate its efficiency in transporting data. The lower the time factor, the more efficient the line, path, or route. See also [surge](#).

**TLB.** See [translation lookaside buffer \(TLB\)](#).

**TMF.** See [HP NonStop™ Transaction Management Facility \(TMF\)](#).

**TNS.** HP computers that support the HP NonStop™ Kernel operating system and that are based on complex instruction-set computing (CISC) technology. TNS processors implement the TNS instruction set. Contrast with [TNS/R](#).

**TNS code segment index.** A value in the range 0 through 31 that indexes a code segment within the current user code, user library, system code, or system library space. This value can be encoded in five bits.

**TNS code space.** One of four addressable collections of TNS object code in a TNS process. They are User Code (UC), User Library (UL), System Code (SC), and System Library (SL). UC and UL exist on a per-process basis. SC and SL exist on a per-node basis.

**TNS instructions.** Stack-oriented, 16-bit machine instructions defined as part of the TNS environment. On TNS systems, TNS instructions are implemented by microcode; on TNS/R systems, TNS instructions are implemented by millicode routines or by translation to an equivalent sequence of RISC instructions.

**TNS library.** A single, optional, TNS-compiled loadfile associated with one or more application loadfiles. If a user library has its own global or static variables, it is called a TNS shared run-time library (TNS SRL). Otherwise it is called a User Library (UL).

**TNS loading.** A task performed at process startup time when executing a TNS object file. This task involves mapping the TNS instructions, procedure entry point (PEP) table, and external entry point (XEP) table from a TNS object file into memory.

**TNS mode.** The operational environment in which TNS instructions execute by inline interpretation. See also [accelerated mode](#) and [TNS/R native mode](#).

**TNS object code.** The TNS instructions that result from processing program source code with a TNS language compiler. TNS object code executes on both TNS and TNS/R systems.

**TNS object file.** The object file created by a TNS compiler. The file contains TNS instructions and other information needed to construct the code spaces and the initial data for a TNS process.

**TNS process.** A process initiated by executing a TNS or accelerated object file. A TNS process, whether accelerated or not, uses TNS register and stack conventions. Contrast with [TNS/R native process](#).

**TNS shared run-time library (TNS SRL).** A shared run-time library (SRL) available to TNS processes in the Open System Services (OSS) environment. A TNS process can have only one TNS SRL. A TNS SRL is implemented as a special user library that allows shared global data.

**TNS signal.** A signal model available to TNS processes in the Guardian environment. |

**TNS stack segment.** See [TNS user data segment](#). |

**TNS system library.** A collection of HP-supplied TNS-compiled routines available to all TNS processes. There is no per-program or per-process customization of this library. All routines are immediately available to a new process; no dynamic loading of code or creation of instance data segments is involved. See also [native system library](#).

**TNS user data segment.** In a TNS process, the segment at virtual address zero; its length is limited to 128 kilobytes. A TNS program's global variables, stack, and 16-bit heap must fit within the first 64 kilobytes. See also [compiler extended-data segment](#). |

**TNS user library.** A user library available to TNS processes in the Guardian environment. |

**TNS/R.** HP computers that support the HP NonStop™ Kernel operating system and that are based on reduced instruction-set computing (RISC) technology. TNS/R processors implement the RISC instruction set and are upwardly compatible with the TNS system-level architecture. Systems with these processors include most of the HP NonStop™ servers. Contrast with [TNS](#).

**TNS/R library.** A TNS/R native-mode library. For a PIC-compiled application, TNS/R libraries can be dynamic-link libraries (DLLs) or hybridized native shared runtime libraries (SRLs). For a non PIC-compiled application, TNS/R libraries can only be native SRLs.

**TNS/R native mode.** The operational environment in which native-compiled RISC instructions execute. See also [accelerated mode](#) and [TNS mode](#).

**TNS/R native process.** A process initiated by executing code that has been compiled directly to RISC instructions, rather than to TNS instructions. Such a process uses RISC register and stack conventions and executes in [TNS/R native mode](#).

**TNS/R native shared run-time library (TNS/R native SRL).** A shared run-time library (SRL) available to TNS/R native processes in both the Guardian and Open System Services (OSS) environments. TNS/R native SRLs can be either public or private. A TNS/R native process can have multiple public SRLs but only one private SRL.

**Token-Ring ServerNet adapter (TRSA).** A ServerNet adapter that provides a single line from an HP NonStop™ S-series server to a token-ring network, allowing the server to act as a station on the ring. The 3862 TRSA can be configured to support network speeds of 4 megabits/second (Mbps) or 16 Mbps, and the media can be either [shielded twisted pair \(STP\)](#) or [unshielded twisted pair \(UTP\)](#).

**topology.** The physical layout of components that define a local area network (LAN), wide area network (WAN), or ServerNet fabric. See also [star topology](#) and [tetrahedral topology](#).

**topology branch.** A processor enclosure and the I/O enclosures attached to it.

**total harmonic distortion (THD).** The ratio, expressed in percent, of the root mean square (rms) value for all harmonics present in the output of a power source to the total rms voltage at the output, for a pure sine-wave output. The lower the THD, the better the power source.

**Total Information Manager (TIM).** The software package for viewing and searching HP NonStop™ S-series hardware and software documentation. You can use TIM software while your computer is connected to a network or on a stand-alone workstation.

**transformer.** Equipment used to step up or step down alternating-current (AC) voltage to meet the specific requirements of the load. A transformer also provides isolation and noise-attenuation properties.

**transient.** A short-duration, high-amplitude impulse that is imposed upon the normal voltage or current.

**translation lookaside buffer (TLB).** A special-purpose cache, part of the RISC processor chip, that is used in quickly translating virtual addresses to physical addresses. This rapid translation is accomplished by remembering and reusing the translations of recently referenced pages.

**Transmission Control Protocol (TCP).** A connection-oriented protocol that provides for the reliable exchange of data between a sending and a receiving system. TCP implements functions corresponding to the Open Systems Interconnection (OSI) reference model Layer 4, the transport layer.

**Transmission Control Protocol/Internet Protocol (TCP/IP).** A set of layered communication protocols for connecting workstations and larger systems in both local area networks (LANs) and wide area networks (WANs). See also [HP NonStop™ TCP/IP](#) and [Parallel Library TCP/IP](#).

**tri-star topology.** A network [topology](#) that uses up to three HP NonStop™ Cluster Switches for each external fabric. External routing is implemented between the three star groups of a ServerNet cluster. (A star group consists of the eight nodes attached to one set of cluster switches.) The star groups are joined by [two-lane links](#). Introduced with the G06.14 release version update (RVU) of the HP NonStop ServerNet Cluster product, the tri-star topology supports up to 24 nodes. See also [split-star topology](#), [star topology](#), and [layered topology](#).

**Trivial File Transfer Protocol (TFTP).** A protocol defined by Request for Comments (RFC) 1350. TFTP is used as a data link control (DLC) and diagnostic task.

**TRSA.** See [Token-Ring ServerNet adapter \(TRSA\)](#).

**TSM.** See [Compaq TSM](#).

**TSM client software.** See [Compaq TSM client software](#).

**TSM EMS Event Viewer Application.** See [Compaq TSM Event Viewer](#).

**TSM Low-Level Link Application.** See [Compaq TSM Low-Level Link](#).

**TSM Notification Director Application.** A component of the Compaq TSM client software. The TSM Notification Director Application receives incident reports from an HP NonStop™ S-series server, displays them, and allows you to take action or forward the incident reports to your service provider for resolution. The TSM Notification Director can be configured to run on a system console at all times, even when other TSM applications are not being used.

**TSM package.** See [Compaq TSM package](#).

**TSM server software.** See [Compaq TSM server software](#).



**TSM Service Application.** See [Compaq TSM Service Application](#).

**two-lane link.** The two single-mode fiber-optic (SMF) ServerNet cables that connect the HP NonStop™ Cluster Switches on the same external fabric (for example, X1, X2, and X3) in a [tri-star topology](#).

**UCME.** See [uncorrectable memory error \(UCME\)](#).

**UID.** A nonnegative integer that uniquely identifies a user within a node.

In the Open System Services (OSS) environment, the UID is the scalar view of the [HP NonStop™ Kernel user ID](#). The UID is used in the OSS environment for functions normally associated with a UNIX user ID.

**unattended site.** A computer environment where there is no operator residing on site and the only access is from a central monitoring station.

**uncorrectable memory error (UCME).** An error caused by incorrect data at a particular memory location. The cause of the error is such that the error is not automatically corrected by the system, and memory replacement is required. Contrast with [correctable memory error \(CME\)](#).

**undefined.** Pertaining to the use of an incorrect value for data or the incorrect behavior of a program for which the ISO/IEC IS 9945-1:1990 standard imposes no portability requirements.

**undervoltage.** A negative change in the amplitude of the voltage.

**unicode.** A 2-octet (2-byte) character code designed to represent more alphabetic and graphic characters than allowed by the ASCII character set. When the first octet of a unicode character is zero, the second octet maps to the ISO 8859-1 (ASCII) character set. Unicode is the standard character set for encoding characters in implementations of the Java language and is the default character code set used by several Microsoft Windows operating systems.

**uninterruptible power supply (UPS).** The equipment used to provide an uninterruptible source of power to connected equipment if a main power outage occurs. The basic components of any UPS system are a rectifier/charger that converts alternating-current (AC) power to direct-current (DC) power, batteries that store the DC power, and an inverter that converts the DC power back into AC power for distribution to the load.

**unitary segment.** See [segment](#).

**unmount.** To make a fileset inaccessible to the users of a node.

**unplanned outage.** Time during which a computer system is not capable of doing useful work because of an unplanned interruption. Unplanned interruptions can include failures caused by faulty hardware, operator error, or disaster.

**unshielded twisted pair (UTP).** A transmission medium consisting of two twisted conductors with no cable shielding. Contrast with [shielded twisted pair \(STP\)](#).

**unspecified.** Pertaining to the use of a correct value for data or the correct behavior of a program for which the ISO/IEC IS 9945-1:1990 standard imposes no portability requirements.

**UPS.** See [uninterruptible power supply \(UPS\)](#).

**upward compatibility.** The ability of a requester to operate with a server of a later revision level. In this case, the requester is upward-compatible with the server and the server is downward-compatible with the requester. Contrast with [downward compatibility](#).

**User Code (UC).** See [TNS code space](#).

**user code.** A logically distinct part of the HP NonStop™ Kernel operating system that consists of the code for user processes.

**user database.** A database within an HP NonStop™ node that contains the user name, user ID, group ID, initial working directory, and initial user program for each user of the node.

**user ID.** The unique identification of a user within a node.

In the Guardian environment, the term “user ID” usually means the structured view of the [HP NonStop™ Kernel user ID](#). In the Open System Services (OSS) environment, the term “user ID” usually means the scalar view of the HP NonStop™ Kernel user ID—a number called the [UID](#).

**User Library (UL).** See [TNS code space](#).

**user library.** A logically distinct part of the HP NonStop™ Kernel operating system that consists of procedures that the operating system can link to a program file at run time.

**user name.** A string that uniquely identifies a user within the user database for a node.

**UTC.** See [Coordinated Universal Time \(UTC\)](#).

**UTP.** See [unshielded twisted pair \(UTP\)](#).

**V.** See [volt \(V\)](#).

**V.35.** The International Telecommunications Union, Telecommunication Standardization Sector (ITU-T) standard for data transmission at 48 kilobits/second over 60 - 108 kilohertz group band circuits. It contains the 34-pin V.34 connector specifications normally implemented on a modular RJ-45 connector. V.35 is the equivalent of Electronics Industry Association (EIA) RS-422/RS-449.

**V AC.** Volts of alternating current.



**vertical tetrahedron.** A topology of HP NonStop™ S-series servers in which the ServerNet connections among the layers of a [cluster switch group](#) form a tetrahedron. See also [tetrahedron](#), [tetrahedral topology](#), and [cluster switch layer](#).

**virtual file system.** In UNIX and Open System Services (OSS), a file system that allows files of a fileset to be distributed across several physical devices.

**volt (V).** The standard unit of measure of the potential difference that is required to move an electric charge.

**volume.** A logical disk, which can be one or two magnetic disk drives or one side of an optical disk cartridge. In HP NonStop™ S-series systems, volumes have names that begin with a dollar sign (\$), such as \$DATA. See also [mirrored disk or volume](#) and [optical disk volume](#).

**WAN.** See [wide area network \(WAN\)](#).

**WANBoot process.** A process provided as part of the wide area network (WAN) subsystem that implements the BOOTP protocol and provides management functions to the WAN subsystem and the WAN products.

**WAN concentrator.** See [ServerNet wide area network \(SWAN\) concentrator](#) and [ServerNet wide area network \(SWAN\) 2 concentrator](#).

**WAN shared driver.** A driver, provided as part of the wide area network (WAN) subsystem, that provides a simplified interface to HP NonStop™ TCP/IP for use by I/O processes. The shared driver interface is similar to that provided by DOIOPLEASE on earlier systems.

**WAN subsystem.** See [wide area network \(WAN\) subsystem](#).

**WAN subsystem manager process.** A process named \$ZZWAN provided as part of the wide area network (WAN) subsystem that starts and manages the WAN subsystem objects, the WAN product process, and device objects. Subsystem Control Facility (SCF) commands are directed to the WAN subsystem manager process for configuring and managing the WAN subsystem and the ServerNet wide area network (SWAN) concentrator.

**WAN Wizard Pro.** A graphical user interface (GUI) that guides you step-by-step through the configuration of wide area network (WAN) and local area network (LAN) software and hardware.

**wide area network (WAN).** A network that operates over a larger geographical area than a [local area network \(LAN\)](#)—typically, an area with a radius greater than one kilometer. The elements of a WAN may be separated by distances great enough to require telephone communications.

**wide area network (WAN) subsystem.** The Subsystem Control Facility (SCF) subsystem for configuration and management of WAN objects in G-series release version updates (RVUs).

**wild-card character.** A character that stands for any possible character or characters in a search string or in a name applying to multiple objects. In Subsystem Control Facility (SCF) object-name templates, two wild-card characters can appear: ? (question mark) for a single character and \* (asterisk) for zero or more consecutive characters. See also [object-name template](#).

**Windows Internet Name Service (WINS).** A name resolution service that resolves Windows NT networking computer names to Internet protocol (IP) addresses in a routed environment. A WINS server handles name registration, queries, and release version updates (RVUs). See also [IP address](#).

**WINS.** See [Windows Internet Name Service \(WINS\)](#).

**work files.** Temporary files created during system generation that serve as storage areas. Work files are useful for debugging purposes after system generation. You can choose to make specified work files permanent.

**working directory.** In the Open System Services (OSS) environment, a directory, associated with a process, that is used in pathname resolution for relative pathnames.

**WORM.** See [write once, read many \(WORM\)](#).

**wormhole routing.** A technique for reducing network latency in a router. Packet bytes are immediately switched to the appropriate output port as soon as they arrive, rather than accumulated in a buffer until an entire packet has been received. Contrast with [store and forward routing](#).

**write once, read many (WORM).** A media storage class in which data, once written, cannot be erased or overwritten.

**wye.** A polyphase electrical supply where the conductors of the source transformer are connected to the terminals in a physical arrangement that resembles the letter Y. Each point of the Y represents the connection for a conductor at high potential. The angle of phase displacement between each point on the Y is 120 degrees. The center point of the Y is the common return point for the neutral conductor.

**wye-delta.** The interconnections between a wye source and a delta load.

**X.21.** A digital signaling interface recommended by the International Telecommunications Union, Telecommunication Standardization Sector (ITU-T) that includes specifications for data terminal equipment/data communications equipment (DTE/DCE) physical interface elements, alignment of call control characters and error checking, elements of the call control phase for circuit switched services, data transfer at or below 9600 bits/second, and test loops.

**X fabric.** The X side of the internal or external ServerNet fabrics. See also [fabric](#), [external ServerNet X or Y fabric](#), and [internal ServerNet X or Y fabric](#).

**XIO.** See [extensible input/output \(XIO\)](#).

**XLLINK.** The linker program invoked during system generation to link accelerated (file code 100) TNS object files to create system code and system library files.

**XO bond.** The bond connection on an isolation transformer, installed between the transformer's neutral XO terminal and ground. This bond is required at North American and British sites to provide an effective return path for any equipment conductor fault current back to the neutral of the isolation transformer.

**Y fabric.** The Y side of the internal or external ServerNet fabrics. See also [fabric](#), [external ServerNet X or Y fabric](#), and [internal ServerNet X or Y fabric](#).

**\$YMIOP.** The name of the maintenance I/O process that is built during system generation and is available at system startup.

**\$ZCNF.** The name of the [configuration utility process](#).

**zero-signal reference.** A connecting point, bus, or conductor used as one side of a signal circuit. Such a reference object might or might not necessarily be designated as a ground. Sometimes referred to as a common circuit.

**\$ZEXP.** The name of the Expand manager process.

**\$ZM<sub>nn</sub>.** The name of the QIO monitor process in processor *nn*.

**\$ZNET.** The name of the Subsystem Control Point (SCP) management process.

**zombie process.** In the Open System Services (OSS) environment, a process that has terminated but is still recorded in system tables.

**zone.** See [cluster switch zone](#).

**\$ZPM.** The name of the [persistence manager process](#).

**ZSCL.** The subsystem identifier for the [ServerNet cluster subsystem](#).

**ZSERVER.** The object file name of the \$ZSVR server process for the labeled-tape subsystem.

**ZSMN.** The subsystem identifier for the [external system area network manager process \(SANMAN\)](#).

**\$ZSVR.** The name of the server process for the labeled-tape subsystem. See also [ZSERVER](#).

**ZSYSCONF subvolume.** The subvolume on the \$SYSTEM disk that contains the system configuration database.

**\$ZTC0.** The default transport-provider process that provides Transmission Control Protocol/Internet Protocol (TCP/IP) services to Open System Services (OSS) `AF_INET` sockets programs.

**\$ZZATM.** The name of the Asynchronous Transfer Mode (ATM) monitor process.

**\$ZZFOX.** The name of the Fiber Optic Extension (FOX) monitor process in the ServerNet/FX adapter subsystem.

**\$ZZKRN.** The name of the [Kernel subsystem manager process](#).

**\$ZZLAN.** The name of the ServerNet LAN Systems Access (SLSA) subsystem manager process that is started by the \$ZZKRN Kernel subsystem manager process and maintained by the \$ZPM persistence manager process. See also [LAN manager \(LANMAN\) process](#).

**\$ZZPAM.** The name of the Port Access Method (PAM) manager process.

**\$ZZSCL.** The name of the [ServerNet cluster monitor process \(SNETMON\)](#).

**\$ZZSMN.** The name of the [external system area network manager process \(SANMAN\)](#).

**\$ZZSTO.** The name of the [storage subsystem manager process](#).

**\$ZZWAN.** The name of the wide area network (WAN) subsystem manager process.

---

---

---

---

# Index

## Numbers

6770 cluster switch [1-38](#)

6780 cluster switch [1-46](#)

## A

Absolute address translation [4-42](#)

Absolute addressing [4-6](#), [4-26](#), [4-30](#)

Absolute memory allocation [4-30](#)

Absolute segment [4-30](#)

Accelerated execution

benefits of [6-2](#)

described [6-70/6-84](#)

Accelerated mode [6-2](#)

return from procedure [6-74](#)

switch to TNS [6-72](#)

Accelerated object code

absence, indication of [4-22](#)

allocation [4-20/4-23](#)

expansion factor [4-22](#)

location in code region [6-70](#)

system code [4-28](#)

Accelerated program file [6-70](#)

Accelerator [6-2](#)

Access validation and translation table (AVTT) [2-14](#)

Access validation and translation (AVT) [2-8](#), [9-8](#), [10-6](#)

Acknowledgment packet [9-6](#)

Adapter, ServerNet [1-6](#), [1-14](#), [10-4](#)

Address

SCSI [10-12](#)

ServerNet [2-6](#), [10-4](#)

Address pointer [6-20](#)

Address space identifier (ASID) [4-38](#)

Address translation [4-36/4-51](#)

global, hit [4-42](#)

user space, TLB hit [4-40](#)

Addressing

64-bit [4-42](#)

absolute [4-6](#), [4-26](#), [4-30](#)

aliased [4-30](#)

byte [3-6](#)

code segment [6-18](#), [11-28](#)

data segment [6-22](#), [6-26](#)

half-segment [6-18](#), [6-22](#)

context-free [4-30](#)

formats, virtual memory [4-8](#)

global [4-42](#)

G-relative [6-34](#)

indirect, data segment [6-20](#)

Kseg0 [4-26](#)

Kseg1 [4-26](#)

physical [4-6](#), [4-26](#), [4-28](#)

process-relative [4-6](#)

P-relative [6-16](#)

relative [4-6/4-24](#)

Addressing mode

G-relative [6-20](#), [6-36](#)

L-minus-relative [6-36](#), [6-50](#)

L-plus-relative [6-36](#), [6-46](#)

SG-relative [6-28](#)

S-minus-relative [6-36](#)

Aliased address [4-30](#)

Aliased segment [4-30](#)

Alignment rules, data [3-2/3-4](#)

Allocation

absolute segments [4-30](#)

accelerated code [4-20/4-23](#)

flat segments [4-24](#)

Kseg0 [4-28](#)

Kseg2 [4-30](#), [4-46](#)

main stack [4-14](#)

regions [4-12/4-21](#)

SC and SCr [4-28](#)

## Allocation (continued)

- shared run-time library (SRL) [4-18](#), [4-24](#)
- system code [4-28](#)
- system data [4-28](#)
- system library [4-16/4-21](#)
  - native mode [4-18](#)
  - TNS mode [4-20](#)
- TNS object code [4-20/4-23](#)
- user code, TNS [4-16](#)
- user data, TNS [4-12](#)
- user library, TNS [4-20](#)
- user space [4-24](#)
- Vseg table [4-28](#)

Appearance side [1-6](#)ASID (address space identifier) [4-38](#)AVT (access validation and translation) [2-8](#), [9-8](#), [10-6](#)AVTT (access validation and translation table) [2-14](#)**B**Backup process [1-10](#)Big-endian [3-3](#), [3-6](#)Block transfer engine (BTE) [2-8](#), [2-10](#), [10-6](#)

## Branch instructions

- forward indirect (BFI) [11-12](#)
- illustrated [6-16](#)
- to subprocedure (BSUB) [6-62](#)

BTE descriptor [2-10](#)BTE (block transfer engine) [2-8](#), [10-6](#)Buffer descriptor [2-10](#)Bulk I/O transfer [10-8](#)Byte addressing [3-6](#)

- code segment [6-18](#), [11-28](#)
- data segment [6-22](#), [6-26](#)
- half-segment [6-18](#), [6-22](#)

**C**

## Cables

- layer-to-layer [1-46](#)
- media [1-6](#)
- ServerNet [1-16](#), [1-40](#), [1-42](#), [1-44](#)
- zone-to-zone [1-48](#)

## Cache

- memory [4-34](#)
- readlink [9-14](#)

Cache hit [4-34](#)Callability attribute, procedure [6-32](#)Callable procedure [6-32](#), [6-42](#), [6-78](#), [6-80](#)Carry (K) register [3-10](#)Cause register [8-4](#)CBA (context-bound address) [4-50](#)CC (Condition Code) [3-9](#)CDB (command data block) [10-16](#)CE (command entry) [10-10](#), [10-12](#)Checkpoint message [1-10](#)

## Cluster

- defined [1-36](#)
- in FOX ring [1-38](#)
- ServerNet [1-38/1-49](#)
- topologies for 6770 switches [1-38](#)
- topologies for 6780 switches [1-46](#)

Cluster switch [1-38/1-49](#)6770 [1-38](#)6780 [1-46](#)Code segment [5-6](#)current [4-12](#), [6-82](#)Command data block (CDB) [10-16](#)Command descriptor block (I/O) [10-12](#)Command entry (CE) [10-10](#), [10-12](#)Communications monitor [10-2](#)Communications work queue [10-18](#)Condition Code (CC) [3-9](#)Context-bound address (CBA) [4-50](#)Context-free addressing [4-30](#)

Controller  
    See ServerNet addressable controller (SAC)  
Core tetrahedron [1-4](#)  
CRU group number [1-28](#)  
CRU (customer-replaceable unit) [1-2](#)  
Current code segment [4-12](#), [6-82](#)  
Customer-replaceable unit (CRU) [1-2](#)

## D

Data alignment rules [3-2/3-4](#)  
Data cache [4-34](#)  
Data segment [5-6](#), [6-20](#)  
Debug stack [4-30](#)  
Decimal arithmetic [3-10](#)  
Direct addressing [6-16](#)  
Direct branch address [6-16](#)  
Direct jump area [6-80](#), [7-16](#)  
Dmap (Debug map) [6-76](#)  
Doubleword [3-4](#)

## E

Enclosure  
    I/O [1-2](#)  
    processor [1-2](#)  
    system [1-6](#)  
Endian convention [3-3](#), [3-6](#)  
Enter\_Priv routine [7-12](#)  
Entry point [6-30](#)  
Environment (ENV) register  
    defined [6-6](#)  
    for procedure call [6-42/6-45](#)  
    in interrupt stack [8-6](#)  
    in RISC register [6-82](#)  
    saved on interrupt [8-6](#)  
Environment, procedure [6-40](#)  
Ethernet port [10-4](#)

Execution mode  
    defined [6-2](#)  
    flag [5-10](#)  
    indicator [8-6](#)  
    transition [5-10](#)  
EXIT instruction  
    accelerated equivalent [6-74](#)  
    deleting parameters with [6-54](#)  
    described [6-40](#)  
    illustrated [6-44](#), [6-54](#)  
    segment identification, use of [6-64](#)  
Exit\_Priv routine [7-12](#)  
Expand  
    network [1-38](#)  
    node [1-38](#)  
Extended data segment  
    See Selectable segment  
External entry point (XEP) table [6-30](#), [6-65/6-70](#)  
External procedure call  
    defined [6-4](#)  
    described [6-64/6-70](#)  
    resolving virtual address of [6-68](#)  
External ServerNet fabric [1-38](#)

## F

Fabric, ServerNet  
    external [1-38](#)  
    internal [1-10](#), [1-38](#)  
Far gateway [7-17](#)  
Far jump [6-80](#)  
Far jump table [6-80](#), [7-16](#)  
Fault tolerance [1-10](#)  
Flag, execution mode [5-10](#)  
Flat segments  
    allocation of [4-24](#)  
    defined [4-10](#)  
Floating-point arithmetic [3-11](#)  
Foreign process data space [4-50](#)  
Four-lane link [1-42](#)



FOX ring [1-38](#)

## Frame

interrupt stack [8-6](#)

origin [7-4](#)

physical memory [4-4](#), [4-36](#), [4-44](#)

pointer [7-4](#)

stack

RISC [7-4](#), [7-12](#)

TNS [6-34](#)

## G

Gateway [6-84](#)

## Gateway table

accelerated mode [6-78](#), [7-14](#)

native mode [7-14](#)

General-purpose register (GPR),

RISC [6-82](#), [7-2](#), [8-6](#)

Global address [4-42](#)

Global area, stack [6-36](#)

## Global data

native process [5-4](#)

system [6-28](#)

TNS process [6-34](#)

Global pointer, native process [7-2](#)

Global variables, process [4-12](#), [6-20](#)

GPR (general-purpose register),

RISC [6-82](#), [7-2](#), [8-6](#)

Group number, CRU [1-28](#)

G-relative address [6-34](#)

G-relative addressing mode [6-20](#), [6-36](#)

G-series RVUs [xv](#)

## H

Half-segment byte addressing [6-18](#), [6-23](#)

Hop count [1-22](#)

## I

IFM (interface monitor) process [10-6](#)

Index register [6-8](#), [6-18](#), [6-24/6-27](#)

## Indexing

code segment [6-18](#)

data segment [6-24/6-27](#)

## Indirect addressing

code segment [6-16](#)

data segment [6-20](#)

Indirect bit [6-16](#)

Indirect branch [6-16](#)

## Instruction

integer [3-9](#)

logical [3-8](#)

Instruction cache [4-34](#)

## Instruction format

doubleword [11-2](#)

memory reference [11-2](#)

Instruction translation lookaside buffer (ITLB) [4-36](#)

INTA (interrupt register) [8-4](#)

Integer instruction [3-9](#)

Interface monitor (IFM) process [10-6](#)

Internal ServerNet fabric [1-38](#)

Interpretation of TNS object code [4-22](#)

Interpreter millicode [6-2](#)

Interprocessor communication (IPC) [9-1](#)

## Interrupt

arithmetic overflow [8-13](#)

correctable memory error (CME) [8-12](#)

dispatcher [8-13](#)

handler [8-2](#), [8-8](#)

instruction breakpoint [8-13](#)

instruction failure [8-12](#)

IPC and I/O [8-13](#)

masking [8-2](#)

memory access breakpoint [8-12](#)

packet [8-4](#), [9-9](#), [9-18](#), [10-14](#)

page fault [8-12](#)

power fail [8-12](#)



## Interrupt (continued)

- power on [8-13](#)
- priority [8-10](#)
- queue [2-14](#), [8-4](#), [9-9](#)
- register (INTA) [8-4](#)
- sampler [8-13](#)
- stack frame [8-6](#)
- stack marker [8-6](#)
- stack overflow [8-13](#)
- time list [8-13](#)
- uncorrectable memory error (UCME) [8-12](#)

Interrupt register (INTA) [8-4](#)IOMF CRU [1-20](#)IPC (interprocessor communication) [9-1](#)ITLB (instruction translation lookaside buffer) [4-36](#)IXA (interrupt exit) instruction [8-8](#)I/O enclosure [1-2](#)I/O multifunction (IOMF) CRU [1-20](#)I/O process [10-2](#)**K**K (Carry) register [3-10](#)

## Kseg0

- access to memory [4-34](#)
- allocation [4-28](#)
- described [4-6](#)
- for physical memory addressing [4-26](#)

## Kseg1

- access to memory [4-32](#)
- described [4-6](#)
- for physical memory addressing [4-26](#)

## Kseg2

- allocation [4-30](#)
- described [4-6](#)
- for global addressing [4-42](#)

**L**L register [6-34](#), [6-36](#), [6-42/6-62](#)Latest user code segment [4-12](#)Layered topology [1-46](#)Layers, switch [1-46](#)LBP instruction [6-18](#)Left-right indicator [11-8](#)

## Link

four-lane [1-42](#)two-lane [1-44](#)Linker process [9-3](#)Linker-listener protocol [9-4](#)Listener process [9-3](#)Little-endian [3-6](#)Local area, stack [6-36](#), [6-46](#)Local variable [6-46](#)Logical instruction [3-8](#)Logical segment [4-10](#)Low-water mark [10-24](#)LWP instruction [6-18](#)LX register [8-6](#)LXi [8-6](#)L-minus-relative addressing mode [6-36](#), [6-50](#)L-plus-relative addressing mode [6-36](#), [6-46](#)**M**

## Main stack

allocation of [4-14](#)described [5-4](#)Mask field [4-40](#)Mask register [8-10](#)Memory access [4-32/4-51](#)Memory cache [4-34](#)Memory descriptor [2-10](#)Memory-exact point [6-76](#)Memory-mapped registers [4-6](#)

**Message**

- checkpoint [1-10](#)

- defined [9-1](#)

- system [9-1](#)

- transfer [9-8](#)

- transfer protocol [9-2](#), [9-6](#)

MFIOB (multifunction I/O board) [1-8](#), [10-4](#)

Millicode [4-18](#), [4-20](#), [4-28](#)

Mirrored volume [1-12](#)

**Mode**

- accelerated

- See Accelerated mode

- addressing

- See Addressing mode

- execution

- See Execution mode

- privileged

- See Privileged mode

- TNS [6-2](#)

Modular ServerNet expansion board (MSEB) [1-6](#)

Module driver [10-6](#)

Monitor, communications [10-2](#)

**Movestep**

- See Right-left indicator

MSEB (modular ServerNet expansion board) [1-6](#)

MSG\_BREAK\_ procedure call [9-4](#), [9-18](#)

MSG\_LINK\_ procedure call [9-4](#)

MSG\_LISTEN\_ procedure call [9-4](#)

MSG\_READCONTROL\_ procedure call [9-4](#), [9-12](#), [9-14](#), [9-18](#)

MSG\_READDATA\_ procedure call [9-4](#), [9-12](#), [9-14](#)

MSG\_REPLY\_ procedure call [9-4](#), [9-18](#)

Multifunction I/O board (MFIOB) [1-8](#), [10-4](#)

**N**

Name space [7-6](#)

**Native**

- object code [5-2](#)

- process [5-2](#)

- system code [4-28](#)

Native I/O controller (NIOC) queue [10-6](#), [10-18](#)

Native mode procedure, calling from TNS [7-6](#)

Nil address [4-30](#), [4-46](#)

NIOC (native I/O controller) queue [10-6](#), [10-18](#)

**Node**

- Expand [1-38](#)

- ServerNet [1-38](#)

Nonaccelerated object code [4-22](#)

Nonprivileged procedure [6-32](#)

Nonprivileged space

- allocation chart [4-24](#)

- defined [4-2](#)

- mapping [4-44](#)

NonStop S7x00 [xv](#)

NonStop Sxx000 [xv](#)

NonStop S-series [xv](#)

Notation, bits [3-3](#)

Null PT [4-48](#)

Null Vseg table [4-44](#)

**O**

Overflow (V) indicator [3-9](#), [8-10](#)

Overflow, floating-point [3-12](#)

**P****Packet**

- acknowledgment [9-6](#)

- interrupt [8-4](#), [9-9](#), [9-18](#), [10-14](#)

- request [2-6](#)

- response [2-6](#)

- ServerNet [2-6](#)

## Packet (continued)

setup [9-2](#)Page [4-4](#), [4-36](#)Page fault [4-36](#)

## Page table (PT)

described [4-44](#)null [4-48](#)

## Parameters, procedure

accessing [6-50](#)deletion of [6-54](#)example [6-52](#)passing [6-48](#)reference [6-48/6-51](#)stacked [6-48/6-56](#)value [6-48/6-51](#)PC (program counter), RISC [6-14](#), [6-76](#)PCAL instruction [6-40/6-43](#)PDST (process data space table) [4-44](#)PEP (procedure entry point) table [6-30](#),  
[6-40/6-43](#)PFS (process file segment) [4-30](#)Physical addressing [4-6](#), [4-26](#), [4-28](#)Pmap address [8-8](#)Pmap table [6-70](#), [6-76](#)PMB (processor and memory board) [1-8](#)PMF CRU [1-2](#), [1-8](#)POP instruction [6-38](#), [11-34](#)Port, router [1-14](#), [1-22](#)Post-pull protocol [9-2](#), [9-10](#)Pre-push buffer [9-10](#), [9-12](#)Pre-push protocol [9-2](#), [9-10](#)Primary cache [4-34](#)Primary process [1-10](#)Privileged mode [6-32](#)accelerated transition [6-78](#)native transition [7-12/7-17](#)Privileged procedure [6-32](#), [6-42](#), [6-80](#),  
[6-84](#), [7-10](#)Privileged space [4-2](#), [4-26/4-31](#)

## Privileged stack

allocation of [4-30](#)defined [5-4](#)Privileged state (RISC) [6-84](#), [7-10](#)

## Procedure

attribute [6-32](#)call instruction [6-40](#)callable [6-32](#), [6-42](#), [6-78](#), [6-80](#)defined [6-30](#)environment [6-40](#)local variables [6-46](#)nonprivileged [6-32](#)

parameters

See Parameters, procedure

privileged [6-32](#), [6-42](#), [6-80](#), [6-84](#), [7-10](#)retrieving returned values [6-60](#)

return

accelerated [6-74](#)nonaccelerated [6-44](#)returning values [6-58](#)

## Procedure call

callable (RISC) [6-84](#), [7-10](#)

external

defined [6-4](#)described [6-64/6-68](#)resolving virtual address of [6-68](#)within segment [6-30/6-62](#)Procedure entry point (PEP) table [6-30](#),  
[6-40/6-43](#)

## Process

code and data allocations [5-6](#)TNS [5-2](#)TNS/R native [5-2](#)Process address space [4-2](#)Process data space table (PDST) [4-44](#)Process file segment (PFS) [4-30](#)

Process identifier (TLBPID)

See TLBPID

Process pair [1-10](#)Processor and memory board (PMB) [1-8](#)

Processor enclosure [1-2](#)  
 Processor multifunction (PMF) CRU [1-2](#), [1-8](#)  
 Processor ServerNet interface [2-2](#), [2-8](#)  
 Process-relative addressing [4-6](#)  
 Program counter (PC), RISC [6-14](#), [6-76](#)  
 Protocol  
     linker-listener [9-4](#)  
     message system [9-6](#)  
     message transfer [9-2](#)  
     post-pull [9-2](#), [9-10](#)  
     pre-push [9-2](#), [9-10](#)  
 PT (page table)  
     See Page table (PT)  
 Pull data [2-8](#), [9-2](#), [9-14](#)  
 Push data [2-8](#), [9-2](#)  
 PUSH instruction [6-38](#), [11-34](#)  
 PX register [6-82](#)  
 P-relative addressing [6-16](#)

## Q

Quadrupleword [3-4](#)  
 Queue  
     interrupt [2-14](#), [8-4](#), [9-9](#), [9-18](#)  
     NIOC (native I/O controller) [10-6](#), [10-18](#)  
     request, I/O [10-6](#), [10-10](#)  
     RX buffers [10-22](#)  
     RX indication [10-23](#)  
     TX command [10-22](#)  
     TX completion [10-22](#)  
     work, communications [10-18](#)  
 Queue services [10-2](#), [10-6](#)

## R

Read transaction, ServerNet [2-6](#)  
 Reading memory [4-34](#)  
 Readlink cache [9-14](#), [9-16](#)  
 Reference parameter [6-48/6-51](#)

Region  
     allocations [4-12/4-21](#)  
     defined [4-4](#)  
     mapping [4-44](#)  
     numbering convention [4-4](#)  
 Register  
     a0 [7-2](#)  
     at [7-2](#)  
     Carry (K) [3-9](#)  
     Cause [8-4](#)  
     general-purpose [6-82](#), [7-2](#), [8-6](#)  
     gp [7-2](#)  
     index [6-8](#), [6-18](#), [6-24/6-27](#)  
     interrupt (INTA) [8-4](#)  
     K (Carry) [3-9](#)  
     k0 [7-2](#)  
     L [6-34](#), [6-36](#), [6-42/6-60](#)  
     LX [8-6](#)  
     mask [8-10](#)  
     memory-mapped [4-6](#)  
     PX [6-82](#)  
     ra [7-2](#)  
     RISC [7-2](#)  
     S [6-34](#), [6-36](#), [6-38](#), [6-42/6-56](#)  
     s0 [7-2](#)  
     sp [7-2](#), [7-4](#)  
     SX [8-6](#)  
     t0 [7-2](#)  
     v0 [7-2](#)  
     \$0 [7-2](#)  
 Register stack  
     accelerated mode [6-82](#)  
     described [6-8/6-12](#)  
     in interrupt stack [8-6](#)  
     simulated [4-46](#)  
     top of [6-38](#)  
 Register stack pointer (RP) [6-8/6-12](#), [6-38](#)  
 Register-exact point [6-76](#)  
 Relative addressing [4-6/4-24](#)

Relative segment number [4-8](#)  
 Request packet [2-6](#)  
 Request queue, I/O [10-6](#), [10-10](#)  
 Response packet [2-6](#)  
 Right-left indicator [11-8](#)  
 RISC environment [8-6](#)  
 RISC processor [6-2](#)  
 RISC registers [7-2](#)  
 RISC word [3-2](#)  
 Rmap (return map) [6-76](#)  
 Router hop [1-22](#)  
 Router port [1-14](#), [1-22](#)  
 Router, ServerNet [1-8](#), [1-10](#)  
 RP address [6-82](#)  
 RP overflow [6-12](#)  
 RP underflow [6-12](#)  
 RP wrap base [6-82](#)  
 RP wrap page [4-46](#), [6-12](#)  
 RP (register stack pointer) [6-8/6-12](#), [6-38](#)  
 RX buffers queue [10-22](#)  
 RX indication queue [10-22](#)  
 RX (receiving) operations, communications [10-20](#)

## S

S register [6-34](#), [6-36](#), [6-38](#), [6-42/6-56](#)  
 S7x00 servers [xv](#)  
 SAC (ServerNet addressable controller) [1-8](#), [1-14](#), [10-4](#)  
 SBB (ServerNet buffer board) [1-18](#), [1-20](#)  
 SBI (ServerNet bus interface) [1-8](#), [1-14](#), [2-2](#), [10-4](#)  
 SCr (system code, RISC) [7-16](#)  
 Scratchpad (SPAD) [4-28](#), [4-46](#)  
 SCSI  
     address [10-12](#)  
     bus [1-12](#)  
     controller [1-8](#), [10-4](#)  
 SEB (ServerNet expansion board) [1-6](#), [1-16](#), [1-18](#)  
 Secondary cache [4-34](#)

Segment [4-30](#)  
     absolute [4-30](#)  
     aliased [4-30](#)  
     code [5-6](#)  
     data [5-6](#), [6-20](#)  
     extended data [4-10](#)  
     flat [4-10](#)  
     latest user code [4-12](#)  
     logical [4-10](#)  
     selectable [4-10](#)  
     sharing [5-8](#)  
     system code [4-28](#)  
     system data  
         See System data segment  
     unitary [4-4](#)  
     user data, TNS [4-12](#), [5-4](#), [5-6](#), [6-4](#), [6-34](#)  
 SEGMENT\_ALLOCATE\_ call [4-10](#)  
 SEGMENT\_USE\_ call [4-10](#)  
 Selectable segment [4-10](#)  
 Serial maintenance bus (SMB) [1-8](#), [10-4](#)  
 ServerNet  
     cables [1-16](#), [1-40](#), [1-42](#), [1-44](#)  
     interface [2-2](#)  
     interrupt [8-4](#)  
     local device [2-8](#)  
     node [1-38](#)  
     processor interface [2-2](#), [2-8](#)  
     remote device [2-8](#)  
 ServerNet adapter [1-6](#), [10-4](#)  
 ServerNet address [2-6](#), [10-4](#)  
 ServerNet addressable controller (SAC) [1-8](#), [1-14](#), [10-4](#)  
 ServerNet buffer board (SBB) [1-18](#), [1-20](#)  
 ServerNet bus interface (SBI) [1-8](#), [1-14](#), [2-2](#), [10-4](#)  
 ServerNet cluster [1-38/1-49](#)  
 ServerNet device [2-2](#), [2-4](#)  
 ServerNet end device [2-2](#)  
 ServerNet expansion board (SEB)  
     See SEB (ServerNet expansion board)

- ServerNet fabric
  - external [1-38](#)
  - internal [1-10](#), [1-14](#), [1-38](#)
- ServerNet ID [2-4](#)
- ServerNet link [1-22/1-24](#)
- ServerNet packet [2-6](#)
- ServerNet router [1-8](#), [1-10](#)
- ServerNet services [2-10](#), [8-4](#), [10-6](#)
- ServerNet subdevice [2-4](#)
- ServerNet transaction [2-6](#)
- Server, defined [1-2](#)
- Service processor (SP) [1-8](#)
- Service side [1-6](#)
- Setup packet [9-2](#)
- SG-relative addressing mode [6-28](#)
- Shared run-time library (SRL), allocation of [4-18](#)
- Sharing segments [5-8](#)
- Shell map [7-6](#)
- Signed number [3-8](#)
- SIV (system interrupt vector) [8-4](#), [8-8](#)
- Size
  - page [4-4](#)
  - region [4-4](#)
  - unitary segment [4-4](#)
- SLr (system library, RISC)
  - code allocation [4-16](#)
  - described [7-14](#)
- SL.0 [4-20](#)
- SMB (serial maintenance bus) [1-8](#), [10-4](#)
- sp register [7-2](#), [7-4](#)
- SP (service processor) [1-8](#)
- Space
  - Kseg0 [4-6](#)
  - Kseg1 [4-6](#)
  - Kseg2 [4-6](#)
  - Kseg3 [4-6](#)
  - nonprivileged [4-2](#)
  - privileged [4-2](#), [4-26/4-31](#)
  - process data [4-2](#)
- Space ID [6-4](#), [6-40](#), [6-64/6-70](#)
- SPAD (scratchpad) [4-28](#), [4-46](#)
- Split-star topology [1-42](#)
- Stack
  - Debug [4-30](#)
  - frame
    - interrupt [8-6](#)
    - RISC [7-4](#), [7-12](#)
    - TNS [6-34/6-67](#)
  - global area of [6-34](#)
  - main [5-4](#)
  - marker [6-40/6-68](#)
    - chain [6-56](#)
    - interrupt [8-6](#)
  - overflow
    - interrupt [8-13](#)
    - of TNS stack [6-42](#)
  - privileged
    - allocation of [4-30](#)
    - defined [5-4](#)
  - register
    - See Register stack
  - switch [7-12](#)
  - tip [7-4](#)
  - TNS [5-4](#), [6-4](#), [6-34](#)
  - TNS data [4-12](#)
  - TNS user data [5-4](#), [6-34](#)
- Stack pointer register [7-4](#)
- Star topology [1-40](#)
- Subdevice, ServerNet [2-4](#)
- Sublocal area, stack [6-34](#), [6-36](#)
- Subprocedure [6-30](#)
- Subprocedure call [6-62](#), [7-4](#)
- Subsystem manager process [10-6](#)
- Switch group [1-46](#)
- Switch layers [1-46](#)
- Switch, cluster
  - 6770 [1-38](#)
  - 6780 [1-46](#)
- SX register [8-6](#)
- Sxx000 servers [xv](#)

System data segment  
     absolute access [4-28](#)  
     described [6-28](#)  
 System enclosure [1-6](#)  
 System global data [6-28](#)  
 System interrupt vector (SIV) [8-4](#), [8-8](#)  
 System library  
     allocation [4-16/4-21](#)  
     native mode [4-18](#)  
     TNS mode [4-20](#)  
     defined [5-8](#)  
     native mode [7-6](#)  
 S-minus-relative addressing mode [6-36](#)  
 S-series servers and systems [xv](#)

## T

Tetra 16 topology [1-34](#)  
 Tetra 8 topology [1-34](#)  
 Tetrahedral topology [1-26/1-36](#)  
 Tetrahedron [1-4](#)  
     defined [1-26](#)  
     vertical [1-46](#)  
 TIB (transfer information block) [2-10](#), [9-8](#)  
 TLB (translation lookaside buffer) [4-36/4-49](#)  
     TLB hit [4-36](#)  
     TLB miss [4-36](#)  
 TLBPID  
     described [4-38/4-43](#)  
     owner array [4-38](#), [4-46](#)  
 TNS  
     compatibility modes [6-2](#)  
     data stack [4-12](#), [6-4](#)  
     environment in RISC registers [6-82](#)  
     environment in RISC registers [8-6](#)  
     mode [6-2](#)  
     object code [4-22](#)  
     object code in accelerated file [6-70](#)  
     stack [5-4](#), [6-34](#)

TNS user code  
     See User Code, TNS  
 TNS user data  
     See User data, TNS  
 TNS user library  
     See User library, TNS  
 TNS word [3-2](#)  
 TNS, TNS/R defined [3-1](#)  
 TNS/R native object code [5-2](#)  
     system code [4-28](#)  
 TNS/R native procedure, calling from TNS [7-6](#)  
 TNS/R native process [5-2](#)  
 topologies  
     star, compared [1-38](#)  
 Topology  
     layered [1-46](#)  
     split-star [1-42](#)  
     star [1-40](#)  
     Tetra 16 [1-34](#)  
     Tetra 8 [1-34](#)  
     tri-star [1-44](#)  
     zone [1-48](#)  
 Top-of-stack  
     memory stack [6-38](#)  
     register stack [6-10](#)  
     TNS user data stack [6-34](#)  
 To-RISC shell [7-6](#)  
 Transaction ID, ServerNet [2-6](#)  
 Transaction type, ServerNet [2-6](#)  
 Transfer information block (TIB) [2-10](#), [9-8](#)  
 Transition, execution mode [5-10](#)  
 Translation  
     of object code [4-22](#)  
     of virtual address [4-36/4-51](#)  
 Translation lookaside buffer (TLB)  
     See TLB (translation lookaside buffer)  
 Tri-star topology [1-44](#)  
 Two-lane link [1-44](#)  
 Two's-complement notation [3-9](#)

TX command queue [10-22](#)  
TX completion queue [10-22](#)  
TX (transmission) operations,  
communications [10-20](#)

## U

UCr code allocation [4-16](#)  
UC.0 [4-20](#), [6-4](#)  
UL.0 [4-20](#)  
Unaliased segment [4-30](#), [4-50](#)  
Underflow, exponent [3-12](#)  
Unitary segment [4-4](#)  
Unsigned number [3-8](#)  
User code, TNS  
    allocation [4-16](#)  
    segment [4-16](#)  
    space (UC) [6-4](#)  
User data, TNS  
    allocation [4-12](#)  
    segment [4-12](#), [5-4](#), [5-6](#), [6-34](#)  
    stack [5-4](#), [6-4](#), [6-34](#)  
User library, TNS  
    allocation [4-20](#)  
    segment [4-16](#)  
    space (UL) [6-4](#)

## V

V (Overflow) indicator [3-9](#), [8-10](#)  
Value parameter [6-48/6-51](#)  
Vertical tetrahedron [1-46](#)  
VFP pointer [7-4](#)  
Virtual frame pointer [7-4](#)  
Virtual memory [4-2](#)  
Virtual page address [4-36](#)  
Virtual page number (VPN) [4-40](#)  
Vseg table  
    allocation [4-28](#)  
    described [4-44](#)  
    null [4-44](#), [4-48](#)

## W

Wakeup prod [10-24](#)  
Word [3-2](#)  
Work queue, communications [10-18](#)  
Write transaction, ServerNet [2-6](#)  
Write-back cache [4-34](#)  
Writing to memory [4-34](#)

## X

XCAL instruction [6-64/6-68](#)  
XEP (external entry point) table [6-30](#),  
[6-65/6-70](#)

## Z

Zone topology [1-48](#)

## Special Characters

\$0 to \$31 registers, RISC [6-82](#), [7-2](#)